The University of Sheffield

Department of Computer Science

SIQXC: Schema Independent Queryable XML Compression for Smartphones

Submitted for the degree of Doctor of Philosophy

(PhD Thesis)

Otlhapile Dinakenyane

July 2014

Supervisor: Dr Siobhán North

ABSTRACT

The explosive growth of XML use over the last decade has led to a lot of research on how to best store and access it. This growth has resulted in XML being described as a de facto standard for storage and exchange of data over the web. However, XML has high redundancy because of its self-describing nature making it verbose. The verbose nature of XML poses a storage problem. This has led to much research devoted to XML compression. It has become of more interest since the use of resource constrained devices is also on the rise. These devices are limited in storage space, processing power and also have finite energy. Therefore, these devices cannot cope with storing and processing large XML documents. XML queryable compression methods could be a solution but none of them has a query processor that runs on such devices. Currently, wireless connections are used to alleviate the problem but they have adverse effects on the battery life. They are therefore not a sustainable solution.

This thesis describes an attempt to address this problem by proposing a queryable compressor (SIQXC) with a query processor that runs in a resource constrained environment thereby lowering wireless connection dependency yet alleviating the storage problem. It applies a novel simple 2 tuple integer encoding system, clustering and gzip. SIQXC achieves an average compression ratio of 70% which is higher than most queryable XML compressors and also supports a wide range of XPATH operators making it competitive approach. It was tested through a practical implementation evaluated against the real data that is usually used for XML benchmarking. The evaluation covered the compression ratio, compression time and query evaluation accuracy and response time. SIQXC allows users to some extent locally store and manipulate the otherwise verbose XML on their Smartphones.

DECLARATION

I declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged. This work contains no material that has been submitted previously, in whole or in part, for the award of any other degree or qualification except as specified.

Otlhapile Dinakenyane

ACKNOWLEDGEMENTS

I would like to thank God who granted me this opportunity and sustained me through this journey.

To my family and friends: thank you for your love and support.

I wish to express my sincere gratitude to my supervisor Dr Siobhán North for her unwavering support. I appreciate the time and advice she offered in this thesis.

LIST OF ABBREVIATION

A-D	Ancestor-Descendant
API	Application Programming Interface
CR	Compression Ratio
CT	Compression Time
DOM	Document Object Model
DTD	Document Type Definition
IR	Information Retrieval
J2ME	Java 2 Micro Edition
LOB	Large Binary Object
P-C	Parent-Child
QEP	Query Execution Plan
RDBMS	Relational Database Management System
SAX	Simple API for XML
Sid	Sub tree Identity
SIQXC	Schema Independent Queryable XML Compressor
SGML	Standard Generalized Markup Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XPATH	XML Path Language
XQuery	XML Query Language
XSLT	XML Style sheet Language Transformation XML

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Introduction

The verbose nature of XML has led to much research devoted to XML compression [Tolani and Haritsa, 2000; Min, Park and Chung, 2003; Leighton, Müldner and Diamond, 2005; Arion et al., 2007; Cheng and Ng, 2004; Wei and Wei, 2012; Rocco, Caverlee and Liu, 2005; Ng et al., 2006; Lin et al., 2005; Wang et al., 2004; Wong, Lam and Shui, 2007; Ferragina et al., 2006; Arroyuelo et al., 2010; League and Eng, 2007; Hariharan and Shankar, 2005; Skibinski and Swacha, 2007; Leighton et al., 2005; Adiego, de la Puente and Navarro, 2004; Cheney, 2001; Li, 2003; Liefke and Suciu, 2000]. This has become more of interest since the use of resource constrained devices is also on the rise. These devices are limited in terms of storage space, processing power and they also have finite energy provided by their lithium battery [Oliver, 2009; Fei, Zhong and Jha, 2008; Hu et al., 2010].

These limitations make it difficult for the resource constrained devices to cope with storing and processing XML given that it is verbose. XML queryable compression methods could be a solution but none of them has a query processor that runs on a resource constrained device. Currently, wireless connections (WI-FI, 3G, and 4G) are being used to alleviate the storage problem by loading only the necessary data when required. However wireless connections drain the finite energy [Oliver, 2009; Chareen et al., 2008, Lindholm, 2009] that these devices depend on, therefore, though it solves the memory problem it is not an ideal solution. The adverse effects that wireless connections have on the battery life cannot be ignored.

This thesis describes an attempt to address this problem by proposing a queryable XML compression system with a query processor that runs on a resource constrained environment thereby lowering wireless connections dependency yet alleviating the storage space problem.

1.2 The Thesis Structure

The thesis is structured in way that lays out the basics about XML leading to a discussion of related work like query processing and optimization, resource constrained devices and XML compression. Having been established from the literature the motivation of this work is presented highlighting the objectives. The last chapters of the thesis focus on the design and the experimental evaluation of SIQXC, the proposed system. The chapters are as follows:

Chapter 1: Introduction. This chapter gives a brief overview of XML and outlines the thesis structure giving a brief summary of what each chapter covers.

Chapter 2: XML Background. The basic syntax of XML, its storage, parsers and query languages are briefly discussed in this chapter to lay a foundation of the broad topic of this thesis; XML.

Chapter 3: Query Processing and Optimization. Ideas from labelling schemes and query processing were used in this work therefore this chapter presents the core operations behind query processing and also reviews the existing optimization schemes.

Chapter 4: Resource Constrained Devices. This chapter discusses the limitations that come with using resource constrained devices. It describes the existing solutions and highlights their limitations.

Chapter 5: XML Compression. This chapter discusses the existing, commonly used XML compression methods. It classifies these compression methods and also highlights their limitations with regards to resource constrained devices.

Chapter 6: Motivation. The motivation chapter defines the problem that this work seeks to address by giving an overview of the motivation, stating the hypothesis, goals and showing the contribution that this work makes.

Chapter 7: Schema Independent Queryable XML Compression. In this chapter an overview of the Schema Independent Queryable Compression method (SIQXC) is made. The chapter presents the components of SIQXC giving a brief outline of each component. A complete system design is also shown in this chapter.

Chapter 8: The Compressor and Decompressor. This chapter explains the two components of SIQXC that run on a resource rich platform; the compressor and decompressor. The processes that are carried out in each component are discussed with examples.

Chapter 9: The Query Processor. In this chapter the two components of the query processors are explained. The supported query types are discussed showing how they are evaluated.

Chapter 10: Experimental Design. This chapter presents the design of experiments that are used to evaluate the functionality and performance of the different components of SIQXC. It also discusses real world datasets that are usually used to evaluate XML applications and the most widely used XML benchmarks. The chapter also covers the possibilities of comparative analysis with existing systems.

Chapter 11: Results and Evaluation. The results from the experiments described in Chapter 10 are presented and evaluated in this chapter. The results describe the compression ratios, compression times, query response times and decompression times of different datasets.

Chapter 12: Conclusion and Future Work. The findings of this research are summarised in this chapter. The chapter also suggests areas of future work based on the limitations discussed under the Results and Evaluation chapter.

1.3 Chapter Summary

This thesis proposes an XML compression system with a query processor that runs on a resource constrained environment to allow the verbose XML to some extent be stored and processed in that environment. The thesis describes the background of XML and discusses research work that has been done that is related to this work. It also outlines the design of the proposed system, the experimental design to test it, results and their evaluation and lastly states the findings and suggests areas of future work from the limitations that the results reveal.

CHAPTER 2

THE XML BACKGROUND

2.1 Introduction

This Chapter provides a background on XML and XML databases. It introduces the XML language and its origin. The chapter discusses basic concepts of XML including its syntax, the subject of validity and well-formedness, parsing and XML query languages emphasising XPATH since it is the most widely used query language. This lays the foundation upon which this research is based.

2.2 Overview

The **E**xtensible **M**arkup **L**anguage (XML) is a self-describing semi structured markup language that is widely used for storing and exchanging data. It has been described as a de facto standard for storage and exchange of data over the web [Sakr, 2009; Ng et al., 2006; Lu and Cheng, 2004; Nicola and Van der Linden, 2005; Weiner, Mathis and Härder, 2008; Su-Cheng et al., 2009; Haw and Lee, 2007; Zhou et al., 2009; Wang et al., 2009; Mlynkova and Necasky, 2009; Grimsmo and Bjørklund, 2010; O'Connor and Roantree, 2010; Zhang and Özsu, 2010; Byun and Park, 2010, Xin, He and Cao, 2010]. XML was derived from SGML in 1996 and was later recommended by W3C in 1998 [W3C, 2010; Tidwell, 2002]. Unlike other markup languages such as HTML its focus is not on the appearance of data but storage and exchange. This semi-structured language is popular for its self-describing nature because it carries semantics about the data it represents.

XML is platform independent [Nicola and Linden, 2005] therefore it provides interoperability between different applications [Gulhane and Ali, 2012; Morgan, 2007]. Furthermore, it is readable by both machines and people [Kirk et al., 2005]. Its syntax is fairly simple and it has an extensible vocabulary where tags are user-defined. These attractive features have led to an increased availability of XML databases spread among different domains ranging from medicine, biology, business and ecommerce.

2.3 Basic XML Syntax

The XML syntax includes elements, attributes, comments and sometimes processing instructions. An element is the basic unit of XML markup. The markup describes the structure of an XML document. In this markup, an element is made up of two matching tags; the opening tag '<>' and closing tag '</>'. Each of these tags encloses a tag name. The tag name is normally a description of the data that is held therein if any. Tags are named following specific rule, for example, the name is case sensitive and cannot start with xml or XML. An element usually contains content enclosed between the opening tag and closing tag as shown in the example below:

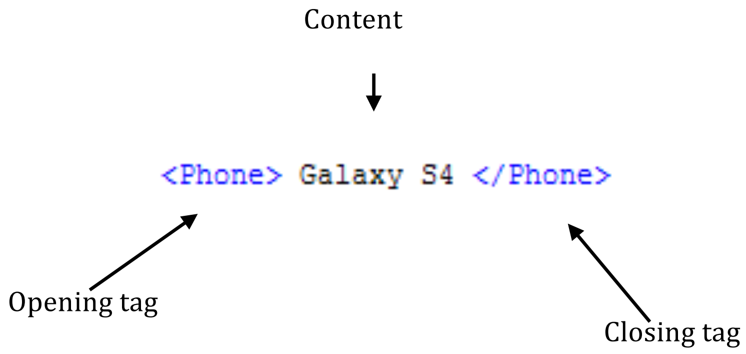


Figure 2.1: An illustration of the XML structure

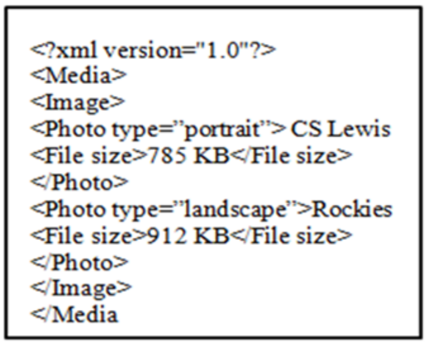
Sometimes an element can be empty. An empty element is represented as a self-closing tag. See the example below:

```
<Image file= "S4.jpeg"/>
```

Figure 2.2: An empty XML tag

Elements can also have attributes that describe their static values. These attributes are enclosed in the opening tag. The syntax of an attribute is such that it has a name and a value. The value is normally surrounded by quotation marks as shown in the *'Image'* example shown above. This element has an attribute name 'file' with a value 'S4.jpeg'.

In addition to having attributes and content an element can contain other elements; child elements. The rule is that all child elements must be closed before the parent's closing tag. The nesting of elements can go on as deep as required resulting in a hierarchical structure (see Figure 2.3 B and 2.3 B below). Figure 2.3 A is a snippet of an generated XML document and Figure 2.3 B shows the elements tree representation of the XML document in Figure 2.3 A.



```
<?xml version="1.0"?>
<Media>
<Image>
<Photo type=”portrait”> CS Lewis
<File size>785 KB</File size>
</Photo>
<Photo type=”landscape”>Rockies
<File size>912 KB</File size>
</Photo>
</Image>
</Media
```

Figure 2.3 A: XML snippet

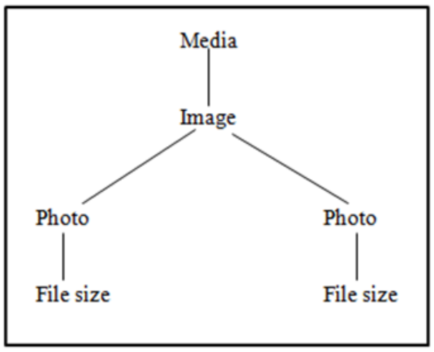
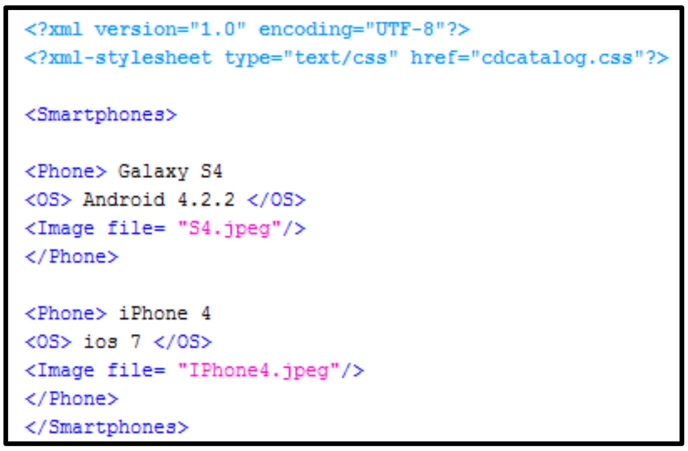


Figure 2.3 B: XML tree

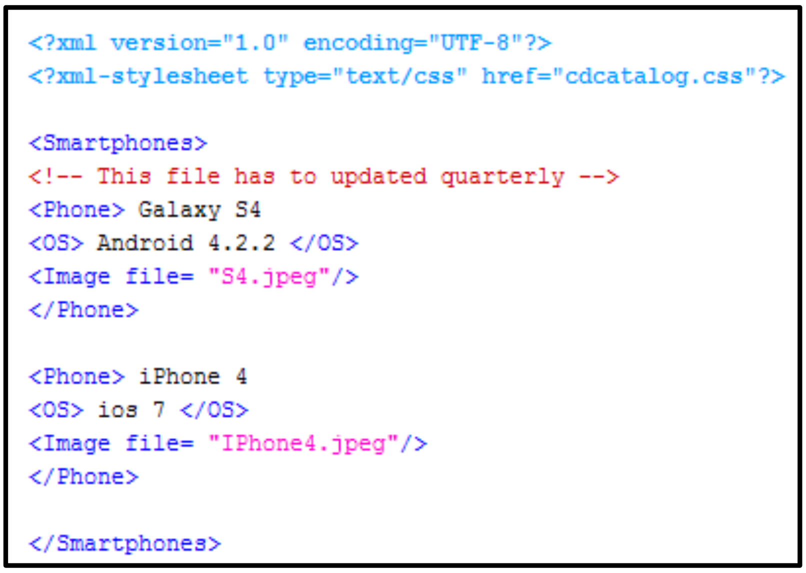
XML documents may contain an optional piece called the prolog which comes before the first element (root element). This is where the XML declaration which indicates the version of XML used in the document is specified. This declaration starts with '<?' and ends with '?>'. It forms part of the important markup called the processing instructions (PI). In addition to the XML declaration, the PI may specify the style sheet that a browser has to comply with in displaying the XML document. Figure 2.4 shows XML document with the PI discussed.



```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="cdcatalog.css"?>

<Smartphones>

<Phone> Galaxy S4
<OS> Android 4.2.2 </OS>
<Image file= "S4.jpeg"/>
</Phone>

<Phone> iPhone 4
<OS> ios 7 </OS>
<Image file= "IPhone4.jpeg"/>
</Phone>
</Smartphones>
```

Figure 2.4: The smartphone XML snippet

An XML document may also have comments that provide further description. They can be added anywhere in the document. Note that, comments are not part of the textual content of the document therefore some XML processors may not even read them. An XML comment begins with '<!--' and ends with '-->'. The XML file in Figure 2.4 is shown below with a comment.



```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="cdcatalog.css"?>

<Smartphones>
<!-- This file has to updated quarterly -->
<Phone> Galaxy S4
<OS> Android 4.2.2 </OS>
<Image file= "S4.jpeg"/>
</Phone>

<Phone> iPhone 4
<OS> ios 7 </OS>
<Image file= "IPhone4.jpeg"/>
</Phone>

</Smartphones>
```

Figure 2.5: The smartphone XML snippet with a comment

2.4 Well Formedness and Validity

Although XML tags are user defined, XML specifies rules that every document must comply with for it to be considered a well formed document. According to these rules, every XML document must have one root element; the first element of the document. Each element should have an opening and closing tag. Tags must be properly nested with no overlapping; tags of subsequent elements must be closed before the parent tag is closed. Furthermore, the opening and closing tags of an element are case sensitive. For most applications it is enough that an XML document is well formed but for some a document has to adhere to additional rules specified in a separate document called a schema.

A schema specifies the structure of an XML document [Papakonstantinou and Vianu, 2000; Fan and Libkin, 2002; Guerrini et al., 2005]. An XML document that has to comply with a certain schema has to be validated against a specific schema to ensure that it meets the requirements outlined therein. There are two widely used schema languages, the **D**ocument **T**ype **D**efinition (DTD) and XML Schema (XSD) [Lee and Chu, 2000; Vitali et al., 2003]. The DTD was the first schema language to be used. In 2004 XSD was passed as a W3C standard and it is the mostly widely used schema. Restrictions included in a schema are; the order in which elements should appear, the number of children an element can have, attributes and sometimes the number of occurrences of an element. This information is helpful in that it maintains data integrity. It can also be exploited to accelerate query execution [Park et al., 2002, Bing-zhang et al., 2010] and to improve the compression ratio of some XML compressors like XAUST [Hariharan and Shankar, 2005], RNGZIP [League and Eng, 2007] and XCQ [Ng et al., 2006].

The XSD is the most widely used schema [Bex, Neven and Van den Bussche, 2004; Bex, Neven and Vansummeren, 2007] because besides offering more restrictions like the use of namespaces and typing, it is written in XML. This allows it to use one tool (validator) for validation and checking the well formedness of a document whereas with a DTD a different tool has to be used.

2.5 XML Storage

An XML file can either be *data centric* or *document centric*. The data centric XML file assumes a highly structured format whereas the document centric file is semi structured text [Sun and Wang, 2011; Noaman and Almansour, 2012]. With a document centric file a few tags are used around text to describe what the text represents. A good example of this is an XML representation of a newspaper article.

Data centric XML files are highly structured and have a high tag to text ratio. This would be best illustrated as an XML representation of a telephone directory. The structured nature of data centric documents makes it easier for them to be processed by machines than the document centric ones which are more human readable.

XML can be considered a database because data can be stored and retrieved from it like other databases. It also shares some other features with the existing database like having a schema and query languages. It also provides interface for programming languages; SAX and DOM (see Section 2.5). It however has some limitations that raised a debate as to whether it should be treated as a database or not [Sun and Wang, 2011; Noaman and Almansour, 2012]. These limitations include lack of security, multi access and recovery [Bourret, 2005; Noaman and Almansour, 2012; Noaman and Almansour, 2012]. As a database, XML can be stored in three different ways described below:

- LOB: In this approach the original XML document is store in a column as a large object.
- Extended relational: This model disintegrates the XML document by shredding it so it can be stored in tables and columns.
- Native: This approach uses a tree structured data model to store XML which is hierarchical by nature.

There are two types of XML databases; Native XML databases and XML Enabled Relational Databases [Haw and Rao, 2005; Bourret, 2005; Noaman and Almansour, 2012]. The first two approaches above are used in the XML Enabled databases and the native approach is used in the Native XML databases as the name suggests. The way data is stored facilitates query processing [Zhang and Ozsu, 2010; Wong, Lam and Shui, 2007] therefore there are costs and benefits of using each of the storage models.

The XML Enabled Relational Databases are relational databases that have the capability of handling XML data by either storing it as a LOB [Simalango, 2010, Tatarinov, et al., 2002] or shredding it into object relational tables and columns [Simalango, 2010]. When stored as LOB, XML can be expensive to manipulate especially if it is a large document because it will have to be loaded in to the memory as a whole.

Shredded XML data on the other hand is easier to manipulate but creates a time overhead because of the mapping and joins that have to be done when executing queries. These processes are necessary because the shredded XML is disintegrated therefore cannot be manipulated in the state it is stored in. XML is shredded to turn its hierarchical model into a tabular one that be stored in relational databases since they are flat. The challenge in these databases is to find a mapping approach [Suei et al., 2009, Haw and Lee, 2010] that preserves semantics at a low cost.

Relational databases are not designed to handle XML data in its native form so manipulating XML in these databases can be costly in terms of efficiency and performance. However they are still needed to handle critical data because they have been used for many years and are very stable. Much research has been done to improve their security, integrity of data, transactions and query optimization which is an advantage over native XML database.

Native XML databases are not as mature as relational databases [Bourret, 2005] nor are they intended to replace them but they handle XML better [Winer and Härder, 2010] because they store it in its hierarchical model. This preserves the semantics held in XML that would otherwise be lost during shredding and normalisation that is required in XML enabled relational databases. Native XML databases store XML as an XML INFOSET, Document Object Model (DOM) or Simple API for XML (SAX) (See the Section 2.5 for the discussion on DOM and SAX).

These models are designed to handle hierarchical data making the cost of query processing low by eliminating the joins and mappings required when using enabled relational databases. Of the three models mentioned, DOM and SAX are the most commonly used. The former supports *breath-first traversal* access pattern through the JAVA getChildNodes method from the Node class whereas the latter demonstrates the *depth-first traversal* access pattern where a pair of *begin*- and end-*events* is generated for each node [Zhang and Özsu, 2009]. The native approach introduces operators that are optimized for tree navigation, deletion, insertion and update [Zhang and Özsu, 2009]. This work assumes a native XML storage approach therefore, the discussion on query processing and optimization in Chapter 3 covers some of these operators and how they have been implemented and optimized.

2.5 XML Parsing

An XML document has to be parsed to be used with any application [Zhao and Bhuyan, 2006, Haw and Rao, 2007]. Parsing XML prepares it to be accessible so that processes like query evaluation can be executed. Programming languages like JAVA provide XML Application Programming Interfaces (API) for XML parsing. There are two main XML APIs; Document Object model (DOM) and Simple API for XML (SAX) [Nicola and John, 2003; Zhao and Bhuyan, 2006; Haw and Rao, 2007]. These APIs define the way in which XML documents are accessed and manipulated; DOM for random access and SAX for serial access. DOM parses an XML document as a whole creating an in memory tree representation [Zhao and Bhuyan, 2006] whereas SAX is an event driven parser that creates tokens or events instead of an in memory tree representation [Nicola and John, 2003, Zhao and Bhuyan, 2006]. With SAX all the processing on a document is done in one cycle whereas in DOM processing can be done as many times as necessary. The type of parser used therefore affects performance [Nicola and John, 2003]. SAX is best for streaming application whereas DOM is suitable for databases [Lam et al., 2006].

2.6 XML Query

An XML query is essentially a path expression or a series of steps which when followed to navigate the XML tree should return the desired node or nodes if they exist [Flesca et al., 2003]. In XML query processing, a user specifies what data they want by giving a path expression that is then followed to extract such data. There are generally two types of XML queries; structural and full text queries [Su-Cheng, 2009]. This classification is discussed further in Chapter 3. Structural queries vary from simple selection to more complex operations such as range queries and structural joins. As stated earlier XML is hierarchical by nature so it can be viewed as a tree with different types of nodes. The nodes in an XML tree include the element nodes, the root node, text nodes (textual content of an element) and attribute nodes. The core operation in XML query processing is identifying the relationships that exists among these nodes [Yun and Chung, 2008; Gou and Chirkova, 2005; Lu et al., 2004; Jiang et al., 2004; Chen et al., 2005; Chen et al., 2006; Xu et al., 2009; Jiang et al., 2009, Lu et al., 2011].

There are three types of relationships; Parent-Child (P-C), Ancestor-Descendent (A-D) and siblings [Yun and Chung, 2007, Su-Cheng; Haw and Lee, 2008, Haw and Lee, 2009]. Assuming the smartphone XML document shown in Figure 2.3 the stated relationships are as follows:

- P-C: This relationship exists between Phone and Image with Phone as a parent
- A-D: OS is a descendant of Smartphone
- Sibling: OS and Image are siblings

XML is queried through an XML query language. The language defines the way queries are processed and operation that are supported. The most widely used XML query languages are the XML Path Language (XPATH) [W3C, 1999] and XML Query Language (XQUERY) [W3C, 2007]. These languages are used to update, retrieve, and delete data.

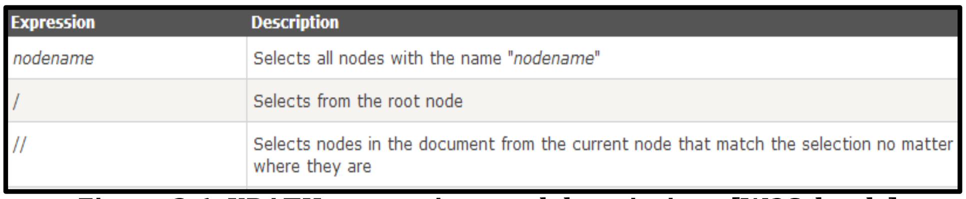
XPATH is however limited in its expressive power therefore does not support complex processes such as transforming the result set like sorting. However, despite its limitations, XPATH is still used by many applications for its simplicity. It was the first query language so it forms the basis for other XML query languages including XQUERY. Both XPATH and XQUERY are World Wide Web Consortium (W3C) recommendations. Other XML query languages recommended by W3C are the XML Pointer Language (XPointer) and the XML Linking Language (XLink). In addition to these there are other languages also used to process XML documents, these include, the XML Style sheet Language Transformation XML (XSLT) , Lorel [Abiteboul et al., 1997], XML QL [Deutsch et al., 1999], XQL [Robie et al., 1999] and QUILT [Chamberlin et al., 2000]. These languages are not discussed further because they are beyond the scope of this work. This work supports XPATH since it is simple and effective and is widely used and supported by applications as stated earlier. This language is therefore discussed further in the following section.

2.6.1 XPATH

An XPATH query is made up of a path expression formulated by nodes separated with backward slashes '/'. Each slash indicate structural relationships among nodes. A path expression that only has singles slashes shows an absolute path of a desired node from the root node. This indicates a P-C relationship. Consider the following examples:

$$\begin{tikzcd}
\text{Smartphones/Phone/OS} & \text{Absolute path} \arrow[l] \\
\text{Or} & \\
\text{Smartphones//OS} & \text{Relative path} \arrow[l]
\end{tikzcd}$$

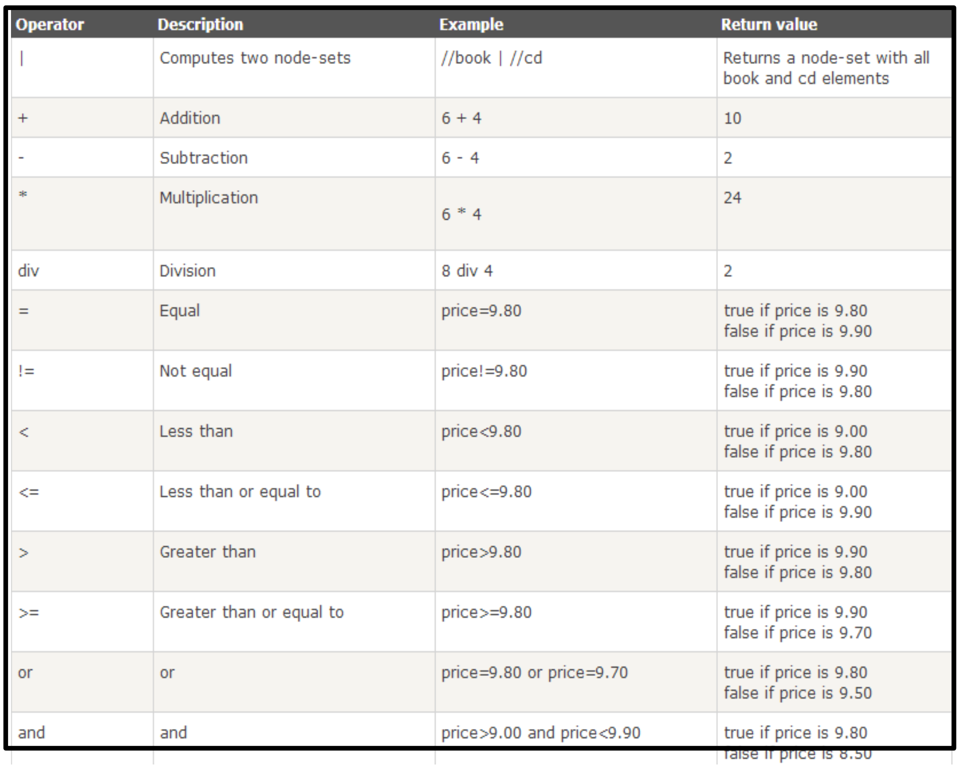
The absolute path is evaluated from the root node while the relative path is evaluated from the context node. The example in Figure 2.6 below shows some of the syntax used in XPATH expressions



Expression	Description
nodename	Selects all nodes with the name "*nodename*"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are

Figure 2.6: XPATH expressions and descriptions [W3Schools]

Optionally a function or predicate is included in this path expression enclosed in square brackets '[]'. A predicate specifies comparative and arithmetical operations to be performed. XPATH supports relational operations like <, >, =, and !=. It also supports logical operations such as 'AND' and 'OR'. The following table indicates a summary of operations supported by XPATH



Operator	Description	Example	Return value
|	Computes two node-sets	//book | //cd	Returns a node-set with all book and cd elements
+	Addition	6 + 4	10
-	Subtraction	6 - 4	2
*	Multiplication	6 * 4	24
div	Division	8 div 4	2
=	Equal	price=9.80	true if price is 9.80 false if price is 9.90
!=	Not equal	price!=9.80	true if price is 9.90 false if price is 9.80
<	Less than	price<9.80	true if price is 9.00 false if price is 9.80
<=	Less than or equal to	price<=9.80	true if price is 9.00 false if price is 9.90
>	Greater than	price>9.80	true if price is 9.90 false if price is 9.80
>=	Greater than or equal to	price>=9.80	true if price is 9.90 false if price is 9.70
or	or	price=9.80 or price=9.70	true if price is 9.80 false if price is 9.50
and	and	price>9.00 and price<9.90	true if price is 9.80 false if price is 8.50

Figure 2.7: XPATH operators with examples [W3Schools]

The XPATH query below requires the OS of a smartphone that has an image with attribute file of value *S4.jpeg.* Assuming the smartphone XML document in Figure 2.5 the result is 'Android'.

/Smartphones/Phone/Image[@file= "S4.jpeg"] /OS

2.7 Chapter Summary

This chapter outlined the basic concepts of XML by discussing its syntax, the way it is stored, parsed and retrieved. It is identified that XML documents can either be document or data centric with data centric documents being more structured. They can be stored in enabled relational databases or in native XML model. Each storage model has cost and benefits therefore careful considerations must be made before choosing the model to use for storing XML data.

It has also been discussed that XML has to be parsed using an API like DOM to be accessible. The parsed XML can be queried and updated through a query language. The most widely used languages are XPATH and XQUERY [Brenes Barahona, 2011]. Query processing in XML heavily depends on the structural relationship among nodes in an XML tree therefore any system that supports query evaluation should ensure that these relationships are retained when the data is stored. Many query optimization schemes and compression methods that support queries somehow retain the relationship among nodes as described in the next chapter.

CHAPTER 3

QUERY PROCESSING AND OPTIMIZATION

3.1 Introduction

XML is hierarchical by nature therefore it can be viewed as a tree with different types of nodes as described in the previous chapter. Querying XML therefore is essentially searching for a path or tree pattern of the specified nodes from this tree. The tree structure of XML makes querying easy in that a specific node can be identified by its relationship with the current node or root node. The relationship among nodes is vital because identifying them is the core operation of query processing in XML [Yun and Chung, 2007, Lu et al., 2004, Jiang et al., 2004, Chen et al., 2005, Chen et al., 2006, Zhu et al., 2008, Xu et al., 2009, Jiang et al., 2009, Lu et al., 2011]. As mentioned in Section 2.7 any system that supports query evaluation has to consider retaining these relationships as much as possible. This chapter discusses the fundamental ideas of query processing and query optimization.

3.2 Query

An XML query in its simplest form is a path expression or a series of steps which when followed to navigate the XML tree return the desired node or nodes if they exist [Flesca et al., 2003]. A query should define the information that the user is looking for, the scope through which it is to be found and the context in which it should be presented [Schlieder, 2002]. The path expression includes steps (node names) separated by a slash (/) and a qualifier or a predicate.

For example: */book/author[@date = "2009"]/title*. The query simply means give all titles of books with an author that has an attribute date of value 2009. This query shows the two basic parts of a query; location path *'/book/author[@date = "2009"]'* and an output expression *'/title'* [Peny et al., 2003]. This is discussed further in Chapter 9.

There are generally two types of XML queries; structural and full text queries [Su-Cheng, 2009; Scioscia and Tinelli, 2011]. Full text queries are keyword based. They do not require a user to know the structure of the XML beforehand. Processing full text queries is mostly Information Retrieval (IR) inclined therefore uses techniques that are borrowed from this discipline. Structural queries are divided further into twig and simple path queries [Haw and Lee, 2009]. A simple path query defines a query based on one element usually from the root element (simple path expression) while a twig query defines a query based on two or more elements (branching path expression). The way a query is processed is dependent on the type of query. The next section discusses query processing.

3.2.1 Query Processing

As mentioned above the type of query determines the way it is processed. The other factor that influences the approach used in query processing is the model in which data is stored. Processing XML queries therefore can either be join-based or navigational [Madria et al., 2007]. If data is highly disintegrated, processing is join-based. This approach mostly followed in enabled relational database. As discussed in Chapter 2 Section 2.5, sometimes data is disintegrated or shredded to be stored in relational databases hence the need to perform several joins during query processing. Joins can also be necessary when dealing with twig queries in native XML storage model. For example, in queries where a disjunction operator is employed joins cannot be avoided. The navigational approach is followed when processing simple path queries on native XML storage model.

Generally query processing involves receiving a query as input, followed by analysis where the syntax and semantics of the query are checked to see if it is valid. If the query is valid redundant parts are removed and a query graph is produced. The query graph is then mapped to a Query Execution Plan (QEP). One of these plans is then executed to get the desired result set [Weiner et al., 2008]. This process is depicted in Figure 3.1 below.

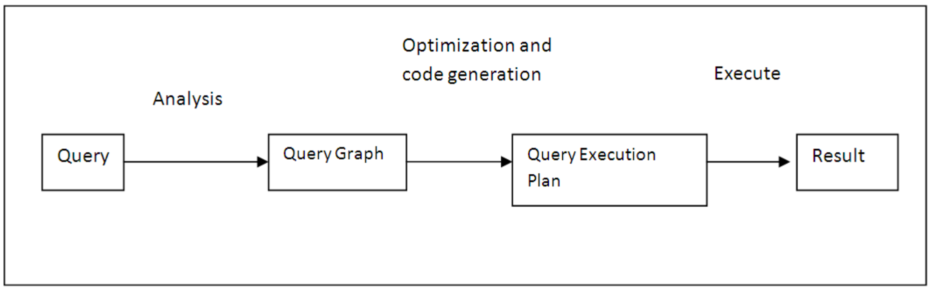


Figure 3.1: Query processing in XML

Several query languages that are mentioned in Section 2.6 are used for query processing to retrieve information; XPATH being the most used because of its simplicity but XQUERY supports more operations. Efficiency is important to every system so in addition to the XML query languages, optimization schemes have been developed to improve query processing performance in terms of response time. Section 3.3 discusses query optimization schemes highlighting their use of structural relationship among nodes to improve query performance.

3.3 Query Optimization

Optimizing query execution in XML is achieved in many ways among which include; **indexing** [Goldman and Widom, 1997; Rizzolo and Mendelzon, 2001; Wang et al., 2003; Chen et al., 2003; Zhang et al., 2006], **labelling schemes** [Weigel et al., 2005; Cohen et al., 2002; O'Neil et al., 2004; Duong and Zhang, 2005; Lee and Hsu, 2004; Yun and Chung, 2008], and **views** [Vidal and Casanova, 2003; Zhou, 2010; Lo et al., 2010; Gao et al., 2010; Liu et al., 2010; Phillips et al., 2010]. Some approaches use a combination of the above methods; the hybrid methods. An example of this is where indexing is used with labelling resulting in a technique called INLAB [Lee and Haw, 2007]. INLAB is an acronym for **IN**dexing and **LAB**elling. In rare cases a schema can also be used sometimes to optimize query processing [Wang et al., 2003]. This is because it is not always advisable to use schema as discussed in Chapter 8.

Optimization schemes have been adopted by some compressors because of their effectiveness in producing structural summaries and efficiency in identifying structural relationship (see Section 4.2.3.2.3).The following sub sections discuss indexing and labelling techniques to illustrate how they improve performance in query processing through structural summaries.

3.3.1 Indexing

Indexing is one of the widely used optimization techniques. Indexing techniques usually group nodes according to their structural similarities [Zhang and Özsu, 2006]. It prunes the XML document resulting in a summary that represents the original XML tree [Chen et al., 2005]. The resulting smaller tree is easier to traverse during query evaluation thereby reducing query execution time. Though they improve query processing efficiency, indexes can consume a lot of space thereby exacerbating the XML storage problem. The compressors that adopt indexing usually compromise compression ratio to increase query support.

An intelligent index consumes less space while optimizing query execution. The structure of an index dictates the type of queries the index can process efficiently because an index suggests the way data is stored which influences the way queries are processed. Researchers over the years developed indexes such as **DataGuides** [Goldman and Widom, 1997], **1-index** [Milo and Suciu, 1999], **ToXin** [Rizzolo and Mendelzon, 2001], **MIS** [Lian et al., 2005], **ViST** [Wang et al., 2003], **FIX** [Zhang et al., 2006], **APEX** [Chung et al., 2002], **A (k)-index** [Kaushik et al., 2002] and **D (k)-index** [Chen et al., 2003] to optimize query processing.

Each of the above indexing techniques is generated using a unique idea that distinguishes it from others. The idea behind the index determines the structure of the index which influences the size of the index and the queries it can process efficiently. This idea also contributes to some of the existing index classifications. The indexing techniques classifications include; Manandhar's accurate and approximate indexes [Manandhar, 2007], Madria et al.'s name, value and path indexes [Madria et al., 2007], Moro et al.'s bisimilarity driven, DataGuides, suffix tree [Moro et al., 2005], Ahn et al.'s sequence, structural, numbering and keyword indexes [Ahn et al., 2005] and lastly Haw and Lee's traditional, similarity based, Forward and backward and advanced indexes [Haw and Lee, 2008].

According to Manandhar accurate indexes are those that reflect the true structure of the source whereas approximate indexes group nodes according to the local structure. An example of accurate index is a DataGuide. APEX and A (k) - index are examples of approximate indexes. This classification is the simplest of those mentioned above. Accurate indexes have an advantage over approximate indexes in that an answer can be derived without traversing the source since these indexes contain full paths from the root for each node. However, accurate indexes suffer from the index size problem which places a high cost on memory. Another shortfall of these indexes is that they do not support branching queries.

Approximate indexes are very effective in pruning the XML tree and give accurate results for short queries. The pruned XML tree is very small because the index does not represent full paths of nodes based on the idea that not all paths are necessary for query evaluation. This assumption is normally based on the workload of XML database which is mined over time. For that reason these indexes fall short in processing long path queries especially if it is a query that has never been executed before. Though accurate indexes consume a lot of space, some compressors still adopt them. The DataGuide is used in XQuec [Arion et al., 2007], a queryable compressor.

Haw and Lee (2008) proposed a classification method that is more detailed than Manandhar's. In their work indexes are classified into four categories. The traditional indexes which are an equivalent of Manandhar‘s accurate indexes. The other three classes indexes are equivalent to Manandhar‘s approximate indexes; these are similarity based indexes which groups nodes according to local similarity, forward and backward indexes which cover the backward and forward path and advanced indexes which are indexes that have been recently proposed like FIX.

The widely used indexes are discussed below outlining the techniques used in creating each index and some of their limitations and strengths:

A DataGuide [Goldman and Widom, 1997] is a path index that shows a structural summary of the database. It gives the actual structure of the database therefore it can be seen as a descriptive schema. According to Chung et al [Chung et al., 2002] and Haw and Rao [Haw and Rao, 2007] the algorithm for creating a DataGuide is like the conversion from non-deterministic finite automaton to deterministic finite automaton [Haw and Rao, 2007; Chung et al., 2002]. This index is an accurate index according to Manandhar's classification discussed above. All the paths that exist in the source are represented but each path is represented only once. It only supports forward traversals from the root so it does not support twig or branching queries, only simple path queries.

A DataGuide can either be minimal or strong [Haw and Rao, 2005]. The minimal DataGuide is compact but has shortcomings. When using the minimal DataGuide the source and the DataGuide has to be traversed simultaneously because the DataGuide does not preserve hierarchical relationships [Haw and Lee, 2008] so it is normally not sufficient to use to process queries. The simultaneous traversal creates a time overhead so a minimal DataGuide is not ideal for query processing especially for large databases. DataGuides are generally not efficient for query optimization because of the space they require for storage.

APEX [Chung et al., 2002] also known as **A**daptive **P**ath Ind**EX** is an approximate index that uses two structures; the Hash tree ($H_{\mathbf{APEX}}$) and the Graph ($G_{\mathbf{APEX}}$) [Haw and Lee, 2008]. The $G_{\mathbf{APEX}}$ keeps the structural summary of paths that are visited frequently. The H_{APEX} on the other hand keeps information about the incoming label path to the respective nodes. Unlike the DataGuide, APEX uses most frequently used paths identified by a sequential data mining [Haw and Lee, 2007] that is performed to formulate the G_{APEX}. An algorithm is run to identify these paths to generate an index. These paths do not necessarily start from the root. Because of its nature of gathering information about frequently used paths, it is sometimes viewed as a workload aware index [Moro et al., 2008]. This index cannot efficiently process new queries because they would not appear in the G_{APEX}. The sequential data mining is expensive [Haw and Rao, 2005] and like the DataGuide, this index does not support branch queries [Haw and Lee, 2008].

The A(k)-index [Kaushik et al., 2002] was made with the assumption that queries are normally short so the index does not necessarily have to represent all the paths that exist in the source or database. Frequently used paths are used to group nodes along them according to local similarity [Haw and Lee, 2007, Wang et al., 2005]. The variable **k** is set by the user. It determines the size of the index and the extent to which it can offer accurate results. The size of **k** is directly proportional to the index size. If **k** is a bigger number then the index will be bigger too. The accuracy of the results is dependent on the length of the query and **k**. This index provides accurate results for queries with paths whose length is less than **k** only [Lian et al., 2005]. Approximation is used to evaluate expressions that have a path longer than **k** but this at times does not give accurate results [Haw and Rao, 2005]. This is an approximate index that only supports simple path queries.

The **F**eature based **I**nde**X**ing, FIX, is based on the spectral graph theory [Zhang et al., 2006]. The XML database is broken down into twig or branched patterns and distinctive features are extracted from each twig pattern to calculate its feature vector based on structural properties. The feature vector is then used as a key or signature for that particular twig. All the keys for different twig patterns are stored in a B^+ tree. When calculating the feature vectors a twig pattern and its value are treated as an object. Incoming queries go through the same process to generate their keys which are then used in the matching process. To evaluate a twig query therefore is mainly looking up for a key from the B^+ tree. In their evaluation of indexes, Haw and Lee [Haw and

Lee, 2008] suggested that FIX has a high pruning power thus efficient in optimizing query processing. This index supports twig queries because of the way it is generated.

3.3.1 Summary

The main aim of indexes is to make data easily accessible by making the XML tree smaller or storing only relevant data to shorten query execution time. The efficiency of an index depends on the data it stores and its structure. In pruning the XML tree relationships are preserved to facilitate efficient query processing.

Some indexes like DataGuides come with a high storage cost because they represent full paths for each node rendering the index inefficient. In the attempt to make an improvement to reduce storage cost, methods which do not represent full paths like A(K) index were developed. The A(K) index however, cannot process queries that have paths that are longer than ***k*** efficiently. Other methods only store frequently used information instead of representing the whole database. These indexes work well until they encounter a new query that is not represented in the index.

When proposing an indexing technique, the challenge is that the index has to be small yet allow a wide range of queries without the need to traverse the original XML tree. Several compressors have adopted indexes to improve accessibility of the compressed data [Ferragina and Manzini, 2005; Arion et al., 2007; Wong et al., 2007; Liao, Hsu and Chen, 2010; Jedidi, Arfaoui and Sassi-Hidri, 2012]

3.3.2 Labelling Schemes

Like indexing a labelling scheme is also a structural summary of an XML tree or the relationships that exist therein [Weigel et al., 2005]. Although they are both structural summaries, labelling schemes are said to be better than indexing techniques in that they require smaller storage space [Li and Ling, 2005; Khaing and Thein, 2006]. The focus in labelling is on individual nodes not paths. Labelling achieves a structural summary by having a unique label for each node in an XML tree with order [O'Connor and Roantree 2010] preserving relationships among nodes. A node label can be either numeric, alphabetic or alphanumeric [O'Connor and Roantree 2010]. Relationships among nodes are necessary for efficient query processing as mentioned earlier in this chapter. Labels are used to identify the relationship between a pair of nodes quickly. As mentioned in the previous chapter, this process is a core operation in query processing [Yun and Chung, 2008; Gou and Chirkova, 2005; Lu et al., 2004; Jiang et al., 2004; Chen et al., 2005; Chen et al., 2006; Xu et al., 2009; Jiang et al., 2009, Lu et al., 2011].

There are many approaches used to label nodes among which include the **Dewey ID** [Tatarinov et al., 2002], **ORDPATH** [O'Neil et al., 2004], **BIRD** [Weigel et al., 2005], **HLSS** [He et al., 2005], **Simple prefix** [Cohen et al., 2002], **Prime Number labelling scheme** [Wu et al., 2004] and **Dynamic interval based labelling scheme** [Yun and Chung, 2008]. These techniques are discussed and evaluated below.

Labelling schemes can be grouped according to their similarities [Wang et al., 2008, Haw and Lee, 2007, Haw and Lee, 2009] as with indexes. There are two basic types of labelling schemes; region encoding, also referred to as range based, and prefix based techniques [Wang et al., 2008, Haw and Lee, 2007]. Prefix labelling schemes are said to be the most diverse labelling scheme class [Haw and Lee, 2009]. This class include the Dewey ID, ORDPATH and Simple Prefix. These labelling schemes use prefix containment to facilitate query evaluation. With prefix labelling schemes relationships are retained by making the parent's label to be part of the child's label. Part of the child's label will always be the parent's label [Härder et al., 2007]. The region encoding schemes on the other hand use intervals or range to define labels [Härder et al., 2007] as demonstrated in the discussion about the BIRD labelling scheme later in this chapter. These methods are effective where there are few updates. A lot of updates would use up the reserved labels quickly leading to the undesirable relabelling. The mentioned labelling schemes are explored in the next sections demonstrating how they facilitate query processing.

The Dewey ID [Tatarinov et al., 2002] is a prefix labelling scheme that is based on the Dewey system used in libraries [Härder et al., 2007, Haw and

Lee, 2009]. In Dewey ID node **x** is a parent of node **y** only and only if it is a prefix of node **y** that means **y**'s label has one delimiter more than **x**. For example if the label for **y** is **2.3.4.1**; **x**'s label will be **2.3.4** if **x** is **y**'s parent. On that account, a label for a specific node allows easy and quick identification of the node's parent or ancestor because these are contained in the child node's label. The parent's label comes before the last delimiter whereas the ancestors' label comes before the last but one delimiter.

Query evaluation with this labelling scheme is therefore done by checking for prefix containment [Gou and Chirkova, 2005]. Dewey ID successfully represents all structural relationships. This accelerates efficient query processing but it uses a lot of space because it represents a full path for each node from the root [Gou and Chirkova, 2005, Haw and Lee, 2009]. It fails to eliminate the redundancy in the original XML tree. To maintain order, relabelling is required when a node is inserted between two existing nodes. The relabelling process is costly and that makes this labelling scheme not ideal where there are frequent updates. Dewey ID, uses a lot of space and does not support updates but it is used for query processing optimization because it effectively stores all structural relationships among nodes.

ORDPATH [O'Neil et al., 2004] is another prefix labelling scheme where a label for a child node is generated by augmenting the parent's label with a component for the child. It uses a similar technique to that of Dewey ID [Haw and Lee, 2009] but ORDPATH reserves even and negative numbers for further insertions to avoid the relabeling required in Dewey ID. This method was proposed to improve the Dewey ID by making it "insert friendly" [O'Neil et al., 2004, Härder et al., 2007]. The reserved labels are used during updates (insertions). As with Dewey ID, ORDPATH preserves structural relationships that are needed for query processing but it is limited by the long labels that it generates because they require a lot of space.

Simple Prefix [Cohen et al., 2002] scheme also belongs to the prefix class. With this labelling scheme a node inherits its parent's label (prefix) and a node's self-label is concatenated to it to form a label for that node. The root label is left empty and the first child's label is assigned **0**, the second and third child's labels are **01** and **011** respectively. Simple prefix is similar to the

Dewey ID scheme; they are both prefix labelling schemes, however, it uses binary representation instead of integers and delimiters. This labelling scheme requires a lot of space for storage [Haw and Lee, 2009] like other prefix labelling schemes.

Besides using binary representation, simple prefix differs from other prefix labelling scheme because the *self-label* comes before the parent's label whereas for Dewey ID and ORDPATH the parent label comes before. Evidently, structural relationships among nodes are retained when labelling an XML tree with this scheme. Query evaluation therefore follows a similar procedure to that discussed with other prefix labelling schemes.

In the prime number labelling scheme [Wu et al., 2004] each node is assigned a prime number and the label for each node is the product of the given prime number and the parent's node label. A relationship exists between two nodes if the child's label in question is divisible by the parent's label therefore prime number labelling scheme. This property makes it a prefix labelling scheme. Unlike other labelling schemes in this class, the prime number labelling scheme is update friendly. Each time a node is inserted it is assigned a new prime number which will then multiplied by the parent's node label, there is no need for recalculations or relabeling. However, like other prefix labelling schemes; it requires a lot more space for storage. Each label is a product of all *self-labels* from the root node, therefore, it takes too much space especially for deep trees. The prime number labelling scheme is effective when dealing with small trees because large trees result in unreasonably large labels. Containment is used for query processing with this labelling scheme because a label contains its parent's label.

In BIRD (**B**alanced **I**ndex based numbering scheme for **R**econstruction and **D**ecision) [Weigel et al., 2005] some labels are left unused for insertion by using a range which determines the number of children a node can have (*weight*) before relabelling is required. For that reason, this labelling scheme is classified as a range encoding scheme. The labelling starts with scanning the database to see how many children a node has [Haw and Lee, 2009]. This information is then used to decide the range. The range must be a number greater than the number of child nodes currently present.

This labelling scheme is ideal for minimal updates particularly insertions. If there are too many updates, reserved places within the range are filled up resulting in the overflow problem [O'Connor and Roantree 2010]. In case of an overflow, the whole tree needs to be relabelled to calculate the new range to create space for further insertions. Given, the tree relations in this labelling scheme, query evaluation involves simple arithmetic operations.

3.3.2 Summary

There are four factors that determine a good labelling scheme. It should be compact, deterministic, dynamic and flexible [Wang et al., 2008, Wu et al., 2004].

- **Compact:** The labelling scheme needs to be compact to use less space when loaded in memory. A compact labelling scheme also means quick query evaluation because the tree representation to be traversed is small but this is only possible if all necessary relationships are retained otherwise it is considered lacking.
- **Flexible:** The flexibility of a labelling scheme makes it portable. A labelling scheme that supports both XPATH and XQUERY is said to be flexible and can be easily adapted into a wide range of XML database systems

- **Deterministic:** A deterministic labelling scheme allows easy identification of relationships. It is vital for evaluating queries. This is achieved well in the prefix based labelling schemes like Dewey ID, ORDPATH and Simple Prefix. BIRD and the prime number labelling scheme also retain structural relationships among nodes but identifying them requires complex arithmetic operations.
- **Dynamic**: A dynamic labelling scheme allows updates without the need to regenerate labels as described in prime number labelling scheme. ORDPATH and BIRD cannot completely avoid relabelling therefore are not dynamic. These labelling schemes reserve nodes to cater for updates but once the overflow is encountered, relabelling is required.

The labelling schemes discussed above satisfy one or two of these factors but none of them satisfies all. They are all not compact enough, with the Dewey ID and Prime Number labelling scheme being the worst. The greatest challenges of labelling schemes are insertions and deletions sometimes referred to as updates. Updates affects order which affects relationships and this often results in the need to regenerate all labels. Regenerating labels is a very expensive process especially for large XML documents. An ideal labelling scheme should avoid relabelling as much as possible. Some labelling schemes like BIRD and ORDPATH attempted to meet this requirement but still fail when all the reserved labels are filled. Labelling schemes are however still better than indexing schemes especially in retaining structural relationships. Ideas from this form of optimization are adopted in SIQXC to prune the XML tree and retain relationships as discussed in Chapter 8.

3.3.3 Hybrid Methods

Some techniques combine labelling and indexing. These methods are referred to as hybrid methods in this work. Hybrid methods include INLAB [Lee and Haw, 2007] and TwigX-Guide [Haw and Lee, 2008]. In the INLAB method, an algorithm called createINLAB is used to label XML nodes and create an index. This algorithm has two components; the XML Encoder which is responsible for labelling and the XML Indexer which creates an index. The label used in the labelling part consists of <self-level: parent> representation. The index is essentially a hash table that stores a node and its associative parent self-attribute. This table facilitates easy relationship identification between a pair of nodes. TwigX-Guide on the other hand uses region encoding and a DataGuide index. It combines the ability of the DataGuide index to evaluate P-C relationships efficiently with the ability of region encoding to evaluate A-D relationships.

This is achieved through two algorithms; *CutMatchMergePath* and *CutMatchMergeTwig* which are used for simple path and twig queries respectively. When a query is received it is evaluated by the *evaluateQuery* method to determine the type of query it is and the appropriate method is invoked. This idea is adopted in SIQXC query processing as discussed in

Chapter 9. If a query is a mixture of A-D and P-C relationships, a structural join is used to get the final result from intermediate results Recall that it mentioned in Section 3.2.1 that joins are necessary for processing branching queries.

3.3.4 Twig Query Processing

It essential to understand twig query processing and the optimization schemes that have been develop specifically for them. Evaluating these queries involves finding all occurrences of the specified tree patterns in the XML tree which is the core operation in XML query processing [Yun and Chung, 2008; Gou and Chirkova, 2005; Lu et al., 2004; Jiang et al., 2004; Chen et al., 2005; Chen et al., 2006; Xu et al., 2009; Jiang et al., 2009, Lu et al., 2011].

Researchers developed different approaches like the Multi Predicate Join Algorithm also known as **MPMGJN** [Zhang et al., 2001], **StackTree** [Khalifa et al., 2002], **TwigStack** [Bruno et al., 2002], **TwigStackList** [Lu et al., 2004], Fast Twig Join algorithm also known as **TJFAST** [Lu et al., 2005], **Twig^{2}Stack** [Chen et al., 2006], **TwigList** [Qin et al., 2007], **TwigBuffer** [Wang et al., 2009], **OTwig** [Liu et al., 2010] and the **Extended XML Tree Pattern Matching** [Lu et al., 2011] to optimize branch query processing. These twig processing methods can be classified as either decompose-match-merge methods or holistic methods depending on the way the process queries.

3.3.4.1 Decompose-Match-Merge methods

A decompose-match-merge method processes queries by first decomposing them into binary structures, matching them and then merging the intermediate results to get the final result. This approach is used in this work to process queries that have conjunction and disjunction operators as discussed in Chapter 9. Storing intermediate results has a high memory cost. Zhang et al. presented the first method for twig pattern matching under this class, MPMGJN [Zhang et al., 2001, Chen et al., 2005]. It is similar to the classical merge-join algorithm used in relational databases (RDBMS). However, this method is much faster than the classical merge-join algorithm.

Its weakness is that it fails to evaluate Parent-Child (P-C) relationships and it also visits too many unnecessary nodes during query evaluation.

StackTree [Khalifa et al., 2002] was proposed to cater for the weakness of MPMGJN. It matches binary structural relationships including the P-C relationship which MPMGJN is unable to evaluate. However, this method suffers from having a large volume of intermediate results which take up a lot of space in memory unnecessarily. It is unnecessary because in most cases, the intermediate results are not part of the final result. Holistic methods were developed to contain this problem.

3.3.4.2 Holistic Methods

Holistic methods avoid generating a large number of intermediate results by processing queries holistically. Bruno et al [Bruno et al., 2002] proposed TwigStack which processes the tree pattern holistically. This method is optimal for Ancestor-Descendent (A-D) edges because it evaluates root to leaf paths against the XML tree. However it is not optimal for P-C relationships because it produces a lot of redundant intermediate results like decompose-match-merge methods in this case. A method can only be considered optimal if all the intermediate results it produces are necessary for the final result [Lu et al., 2005].

Another holistic method called TwigStackList was presented by Lu et al [Lu et al., 2004] designed to meet deficiencies in TwigStack. TwigStackList is optimal for P-C relationships or non-branching edges. It achieves this by employing a look-ahead technique where some elements are cached in memory as they are being read from the input. Other holistic methods are the TJFAST [Lu et al., 2005] and TwigList [Qin et al., 2007]. The former exploits the Dewey ID encoding system by using only leaf nodes to evaluate a query because the leaf nodes labelled using the Dewey ID as discussed in Section 3.3.2.1 carry information about every node in the lineage of the given node. TJFAST therefore reduces the I/O cost. The latter was presented as an improvement on the TwigStackList. Unlike TwigStackList, TwigList uses lists instead of stacks because they are a simpler data structure to work with making TwigList more efficient than TwigStackList.

3.3.4 Summary

Twig pattern matching can either be done holistically or by decomposition. The challenge in twig query processing is to find a method that can evaluate all relationships among nodes effectively without yielding unwanted intermediate results. Holistic methods successfully eliminate intermediate results when evaluating some relationships (A-D). Intermediate results are not only costly on memory but also on processing therefore should be avoided as much as possible

3.3.5 Views

Query optimization can also be achieved through the use of views [Vidal and Casanova, 2003; Zhou, 2010; Lo et al., 2010; Gao et al., 2010; Liu et al., 2010; Phillips et al., 2010]. A view, in traditional relational databases is defined as a stored query whereas in XML databases it is a virtual XML document [Roantree et al., 2007]. It can be an XML database fragments, node references, data values and or full paths [Roantree et al., 2007]. They are derived from or modelled by results from queries already asked or just selected queries [Mandhani and Sucui, 2005; Gire and Idabal, 2008].

When using views, queries are evaluated against a view instead of the original XML tree and since a view is just a subset of this tree, processing time is significantly shortened. Query processing involves checking for containment, rewriting the query or restructuring it and checking for equivalence. In a case where a query cannot be evaluated against a view because equivalence does not exist, it is evaluated against the original XML tree.

3.4 Chapter Summary

The way data is stored affects the way it is retrieved. It also affects query processing time. Disintegrated data results in high use of joins which involves intermediate results that not only use a lot of space but increase query response time. Joins are not necessary when data is stored as a Large Binary Object (LOB) and when XML is stored in a native XML database. The native storage assumes an XML tree that allows easy and fast manipulation of XML. The XML tree can sometimes be too big using a lot of memory thereby increasing query response time. For this reason, optimization techniques have been put in place to provide a smaller, summarised representation of the original tree that allows queries to be evaluated efficiently.

Most optimization techniques like compression (see Chapter 5), reduce the size of the XML tree; however, optimization techniques are focused on easy and quick access of nodes rather than storage cost. This means that with optimization techniques sometimes storage cost can be compromised for a better query response time. Some optimization techniques have been incorporated in compression methods for their ability to prune the XML tree especially the labelling schemes and indexes. This work adopts ideas from query optimization schemes for both pruning and processing queries as shown in Chapter 8**.**

CHAPTER 4

RESOURCE CONSTRAINED DEVICES

4.1 Introduction

The increased processing capabilities of smartphones have led to users depending on them to carry out more tasks than ever before [Hu et al., 2010; Kim, Agrawal and Ungureanu, 2012]. They have pervaded our daily lives [Verbelen et al., 2013]. Smartphones are now used for creating documents, accessing emails, playing multimedia and retrieving information from servers via wireless connections [Whang et al., 2009 Portokalidis et al., 2010] like 3G, 4G and Wi-Fi. They are classified as pervasive and ubiquitous devices [Chareen et al., 2008] because they can be carried everywhere giving users access to their data at any given time where wireless connections are available. Ubiquitous as they are, these devices are resource-restrained; they have limited memory, provide finite power from a lithium battery and relatively slow processor [Oliver, 2009; Fei, Zhong and Jha, 2008; Hu et al., 2010].

This led to a few XML databases developed especially for resource restrained devices. As mentioned in Chapter 5 XML has a repetitive structure therefore verbose [Augeri et al., 2007]. This verbosity nature of XML results in very large files [Cheney, 2006, Arion et al., 2007; Maneth et al., 2008; Zhou et al., 2010; Ng et al., 2006; Arion et al., 2004] that require a lot of space to be stored. It is a problem even for the resource rich devices and more so for resource constrained devices. Storing and querying XML in these devices is a challenge. This led to a lot of research that investigates alternative ways of making XML accessible on limited devices.

This chapter studies different approaches that are currently in place to minimize the storage and processing cost. Some solutions that are discussed in this chapter are not for XML but are worth mentioning because they were developed to address the problems that XML also faces. It studies these approaches highlighting their limitations showing the need to design a better solution.

4.2 Databases for Limited Devices

Many databases have been developed specifically for resource constrained devices. However, these databases have been developed for relational databases not XML databases. Examples of these databases include; **IBM DB2 Everyplace** [Karlsson et al., 2001], **Oracle 10g Lite** [Oracle, 2006], **Oracle Berkeley DB** [Seltzer and Oracle, 2007] and **Microsoft SQL Server Compact Edition** [Dhingra and Swanson, 2007]. In addition to these, there are also research prototypes; **TinyDB** [Madden et al., 2005], **PicoDBMS** [Bobineau et al., 2001], **Odysseus/Mobile** [Whang, 2005; Whang 2007], **mobile database for JAVA phones** [Lu and Cheng, 2004], **XMLDB for embedded systems** [Hoque et al., 2007] and **MonetDB/XQUERY** [Boncz et al., 2006]. Most of these databases are commercial.

The mentioned databases were not developed for smartphones only but for other limited devices like smartcards, sensors and PDAs. The TinyDB was developed at the University of California Berkeley for sensors [Madden et al., 2007] whereas the PicoDBMS is for smartcards [Bobineau et al., 2000]. Only a few of the listed databases were developed for XML. MonetDB/XQUERY for example, is a relational database extended to handle XML data (XML enabled relational database) by supporting XQUERY [Whang et al., 2009]. At the time of writing the author is only aware of two databases that were specifically developed for XML in resource constrained devices; XMLDB and mobile database for JAVA phones. These two databases are briefly discussed below.

4.2.1 Database for JAVA Phones

Lu and Cheng [Lu and Cheng, 2004] used JAVA 2 Micro Edition (J2ME) and Extensible Stylesheet Language Transformation (XSLT) to design a mobile XML database for mobile phones. Their database management system known as DBEngine controls every process run in this database. It has two basic components; tiny-XSLT and kDOM. These are subsets of the standard recommendation of XSLT and DOM respectively. They are both modified to provide only the necessary functions for resource constrained devices.

Tiny-XSLT is used to define rules for manipulating data on the database and kDOM is a parse for J2ME. Database functions supported by the DBEngine include query (data extraction), update, insertion and deletion. Only a specific portion of the database is loaded into memory as and when needed during query execution. However, only a small portion of the database can be loaded so processing range queries poses a challenge because they requires a large portion of the database to be available depending on the range specified in the query. This database does not support the widely used XPATH and XQUERY therefore is limited to XSLT users.

4.2.2 XMLDB for Embedded Systems

Another database developed specifically for XML is the lightweight XMLDB [Hoque et al., 2007]. This database was developed for embedded systems by using J2ME and defining processing rules with an XPATH parser. Like the database for JAVA, it supports functions like insertion, deletion, updating and retrieval. During query processing, data is loaded via the Internet. In this process a mobile phone sends a request to the network. Once the request is accepted the query is processed sending the result set back to the phone. The code that has been used to develop this database is minimal so it can easily be run on a limited device. This database has been tested on a Nokia N73 during development [Hoque et al., 2007] but it is not limited to this platform.

XMLDB's dependency on the wireless connections limits it to be only useful in areas where the Internet is available. The availability of Internet is not dependable in some places and situations as discussed in Section 4.3 later in this chapter

4.2.3 Platform Dependence

Platform dependence is an important subject that needs to be addressed in development especially for applications for smartphones since there are several possibilities. Smartphone platforms include Android, BlackBerry, iPhone (iOS), Symbian, Windows Mobile [Oliver, 2009] and are likely to increase in the future. Application developments for these devices therefore need to support heterogeneous environments [Grønli, Hansen and Ghinea, 2010]. Android and iOS are the most widely used platforms [Goadrich and Rogers, 2011] with Android being the emerging leading platform [Hu et al., 2010].

The iOS only supports Objective-C and its SDK is limited, therefore it provides insufficient functionality for developers. Both Blackberry and Android support JAVA but the use of the Android platform is more than that of the Blackberry platform. The Symbian platform which runs in Nokia phones like the one mentioned in section 4.2.2 supports a wide range of languages among which are C++, JAVA, Python and Perl. However, the use of Nokia phones has been declining recently. The Windows Mobile application development is mainly based on .NET. Android was used in this work because, besides being one of the emerging leading platforms, it supports JAVA.

4.2.4 Summary

Evidently there has not been much research done on XML databases for limited devices yet. However, there are many challenges in this area especially those posed by the verbose nature of XML. The XML databases discussed above manipulate XML by partially loading a small footprint into a resource constrained device. XMLDB relies heavily on the Internet to achieve this. Using wireless brings about some challenges as discussed in Section 4.3

below. The existing solutions have limitations therefore there is need to develop an alternative that would attempt to alleviate them by reducing the wireless connection dependecy.

4.3 The use of Wireless connection

As mention earlier in this Chapter some databases like XMLDB depend heavily on wireless connections to load a small portion of the database as and when needed during query processing. This is because databases developed for resource constrained devices should have a small footprint [Lu et al., 2004, Whang et al., 2009] to accommodate their memory limitations. The database footprint can be as small as 350 kilobytes [Ortiz, 2000]. With this solution, most of the data resides in the server and the smartphone acts as a client. The necessary modules are loaded through the use of a wireless connection. However, wireless connections are not ubiquitous [Satyanarayanan, 1996, Riva and Kangasharju, 2008, Lindholm, 2009]. For that reason a lot of disconnections can be experienced while carrying out database transactions.

Disconnections may be as a result of lack of coverage, radio interference and hand-off between cellular base stations. Hand-offs cause delays during query evaluation because a phone has to acquire a new IP address each time it switches hosts. The delays waste time and this may render a good system inefficient because of the prolonged query response time. A database that uses wireless connections should have some contingency measures in place to deal with issues that come with disconnections. It should be able to rollback unfinished updates or pause and resume when the connection is regained. If the battery runs out the database should be able to self-maintain. Using wireless connections therefore requires a lot from a resource constrained device.

4.3.1 Limited Power Supply

Using the wireless interface is costly on the lithium battery especially during data transmission [Chareen et al., 2008, Lindholm, 2009; Portokalidis et al., 2010; Fei, Zhong and Jha, 2008]. This lithium battery provides finite power supply for Smartphones. Energy consumption should be minimized as much as possible when using these devices. There have been proposals like Power Saving Mode (PMS) introduced to manage energy consumption. PSM is an IEEE 802.11 standard which was introduced in 1997 [Lindholm, 2009]. It saves energy by using three states; awake, sleep and off. The phone is awake when it is actively being used and it is in the sleep mode if it is on standby. The off state means the phone is switched off. It should be noted that this approach only saves power when the phone is sleeping; it cannot save power while the phone is awake which does not solve the wireless connection energy consumption problem.

Other ways of saving energy are the Bounded Slowdown Protocol [Krashinsky and Balakrishnan, 2005] and Smart Power Saving Mode [Qiao and Shin, 2005]. Bounded Slowdown Protocol saves energy by lowering the frequency of nodes listening to the Access Point whereas Smart Power Saving Mode estimates the ultimate sleep time of a smartphone. Lindholm [Lindholm, 2009] states that the energy consumption problem is not likely to disappear even with all these proposals in place. This claim is supported by Cuervo et al. who states that the battery technology is one of the greatest obstacles for smartphones and their growth, and that the technology trends indicate that this problem is here to stay [Cuervo et al., 2010].

In an attempt to provide a solution this problem, developers minimized the functionality of systems that run on smartphones to exclude complex computations that may require a lot of power [Hoque et al., 2007]. One way of doing this is by minimizing the transfer of data between the server and the client since wireless transmissions consume a lot of energy. Minimizing these transfers would mean either having a lot of data on the client side (smartphone in this case) or running a few transactions. Limiting the number of transactions defeats the purpose of using wireless connections as a solution to the XML verbosity problem.

4.3.2 Data integrity and Security

The disconnections and power consumption problems discussed above may result in some inconsistencies in the data held in the phone and the server; therefore, such systems require other functions to synchronize data to maintain data integrity. Synchronization can be achieved by having a simple log with timestamps but it is often complex when dealing with interrupted transactions. Nonetheless, researchers proposed synchronization algorithms to address this problem. One such algorithm is Choi et al.'s that is based on Message Digest (SAMD) [Choi et al 2009]. These solutions require more space especially in cases where a log has to be kept. Synchronization is an extra function that would otherwise not be needed if the whole database or most of it somehow resided in the phone. This shows that there in need for a solution that does not use wireless connections to lower integrity issues.

Another problem that comes with using wireless connections is security. If the Internet is not being used, a password is enough security but using wireless connection increases the number of potential attacks. There is a growing incentive to attack phone because they are now being used for commercial transactions such as online shopping and even banking [Portokalidis et al., 2010]. Therefore, extra resources have to be spent to ensure security when dealing with critical or sensitive data.

4.3.3 Summary

Connecting a resource-restrained device to a resource-rich backend device wirelessly provides a reasonable solution to the memory limitation problem but comes at a cost from a different perspective. There are many complications that come with the use of wireless connections as a solution. These complications are described below.

- **Disconnections**: as discussed earlier in this chapter there are many disconnections that may be experienced when using wireless connections on smartphones. This may be as a result of lack of coverage, radio interference and hand-off between cellular base stations. Disconnections mostly occur because of lack of coverage. Coverage is may not be available while travelling between some places especially in developing countries. In cases of natural disasters such as the 'tsunami' and terrorist attacks; the wireless infrastructure might get affected disrupting the availability of wireless communication [Satyanarayanan et al., 2013]. Wireless is completely unavailable in most aeroplanes therefore users cannot access their data remotely while travelling if it is not stored in the phone.

- **High power consumption**: Wireless connections consume a lot of energy especially during disconnections and hand-offs because the phone continually searches for the nearest base station to connect to. This process uses excessive amount of power adversely affecting the battery life. A lot of energy is also consumed during data transmissions between the smartphone and the server.
- **Exacerbated data integrity problems**: wireless connections allow the same data to be accessed by different clients concurrently. Accessing data concurrently may result in conflicts because of the inconsistencies in the data that is held in different clients thus compromising data integrity. This can happen when data is updated in one or more clients. Synchronization measures have been put in place to reduce the problem however even synchronization comes at a cost.
- **Security**: a system that uses wireless connection is prone to more security attacks than a standalone system. This is a major concern considering that cyber-attacks in the past few years show that it is not just a hypothetical possibility [Satyanarayanan et al., 2013]. Complex security algorithms or some form of cryptography must be incorporated into these systems to protect data. Increasing the number of algorithms consumes space which is one of the limited resources on smartphones. Running the system algorithm also consumes energy which is finite. This makes wireless connection a less feasible solution.

4.4 Chapter Summary

Despite the increasing hardware capabilities, smartphones are resource-restrained and cannot compete with their desktop and server counterparts [Verbelen et al., 2013]. Wireless connections allow data transmission between them and a resource rich server. This allows the smartphone to only store necessary information at a given time reducing the memory cost that would otherwise come with the verbose XML.

Although wireless connections have become helpful in reducing the memory limitations of a smartphone, it comes as a trade-off for power, security and data integrity. They are not always available so high dependency on them can be tragic for critical systems. Physical proximity is a precious attribute [Satyanarayanan et al., 2013] in such cases. It makes a system less vulnerable.

A better solution therefore should be focused more on increasing the physical proximity of data than promoting remote access. This can be achieved by making the XML tree small enough to some extent to fit in the limited memory of a smartphone. The need to reduce the size of an XML tree has led to the much research on XML compression. This is explored further in the next chapter.

Chapter 5

XML COMPRESSION METHODS

5.1 Introduction

As mentioned Chapter 2 XML is a self-describing language. It holds semantics which denote the descriptions of the data held between the opening and closing tags of an element. The self-describing nature of XML is repetitive resulting in large documents with a high ratio of redundancy [Cheney, 2006, Arion et al., 2007; Maneth et al., 2008; Zhou et al., 2010; Ng et al., 2006; Arion et al., 2004;]. XML is therefore notoriously verbose [Augeri et al., 2007; Gulhane and Ali, 2013]. The verbose nature of XML substantially increases the cost of exchanging, storing and processing XML documents. This poses an even greater challenge on limited devices like smartphones.

Generally, XML documents are larger than other files of a different specification holding the same content but the XML data continues to proliferate on the web today thus the need for compression. Compression significantly reduces the disk space required to store the data, saves bandwidth during data exchange and in many cases improves the XML processing performance. As a result many XML specific compressors have been developed [Tolani and Haritsa, 2002; Min et al., 2003; Liefke and Suciu, 2000; Cheney, 2001; Min et al., 2003; Cheng and Ng, 2004; Lin et al., 2005; Leighton et al., 2005; Ng et al., 2006; League and Eng, 2007; Skibinski and Swacha, 2007]. The W3C also formed the Efficient XML Interchange Working Group (EXIWG) that specified an XML binary format to address the storage problem [W3C, n.d].

Compression can either be lossy or lossless. The lossy compressors achieve a better compression by losing some information during the compression process [Salomon, 2004; Pena, 2013]. Decoding the compressed files created through such compression does not yield the exact replica of the original source file. The lossless compression on the other hand preserves all the information during compression. This is mandatory when dealing with text [Pena, 2013]. This work discusses lossless compression because XML is essentially text.

XML is represented as text therefore it is sometimes compressed using general text compressors like gzip [Deutsch, 1996]. However, these traditional compressors are not efficient in compressing XML because they do not take its structure into consideration during compression. The structure of XML has a lot of redundancies. Considering the repetitive structure of XML during compression can improve the compression ratio if the compressor exploits it to remove repetitive structures. Unlike general text compressors, some compressors consider the structure of XML. This has led to the first classification of XML compressors*; XML blind* and *XML conscious* compressors.

XML blind compressors compress XML like a text document whereas *XML conscious* compressors take the structure of XML into account, harnessing it for better compression. As a result XML conscious compressors usually compress data better than the XML blind compressors. The next section briefly discusses the XML blind compressors showing their classification and highlights the main things that differentiate the sub classes.

5.2 XML Blind Compressors

XML *blind* compressors are general text compressors that compress XML without considering its structure. This is a logical thing to do because XML is represented as text. Consequently, XML blind compressors do not exploit the repetitiveness of XML to reduce the redundancy therefore the compression ratio is compromised. These compressors can either be statistical [Knuth, 1985; Brisaboa et al., 2003; Ryabko and Rissanen, 2003; Witten, Neal and Cleary, 1987] or dictionary based [Williams, 1991; Ziv and Lempel, 1977; Ziv and Lempel, 1978]. Figure 5.1 shows this classification.

The statistical compressors compress data by assigning code words to each source symbol. The length of the code word is dependent on the probability of the source symbol within the document. Shorter code words are assigned to source symbols that appear more frequently than others thereby achieving compression. These compressors include the Huffman [Knuth, 1985], dense and arithmetic codes [Witten, Neal and Cleary, 1987; Brisaboa et al., 2003].

The dictionary-based methods on the other hand compress data by forming a dictionary of substrings to which fixed length pointers are linked. Long sequences in this family of compressors are represented by a shorter pointer. Grammar based compressors and Lempel Ziv compressors belong to this family [Ziv and Lempel, 1977; Ziv and Lempel, 1978; Williams, 1991; Adiego, Navarro and others, 2007]. Gzip is a hybrid of the statistical and dictionary compressors since it is a combination of Huffman coding which is statistical and LZ77 which belongs to the Lempel Ziv family; a dictionary based compressor (see Chapter 8 for more on gzip). As a result gzip is widely used for both compressing text and XML. Some XML conscious compressors also use gzip as the back end compressor including this work. It is widely used because it is open source, it has a good compression rate (40-50%) and it does not require the knowledge of the structure of thedocument it compresses [Nair, 2002].

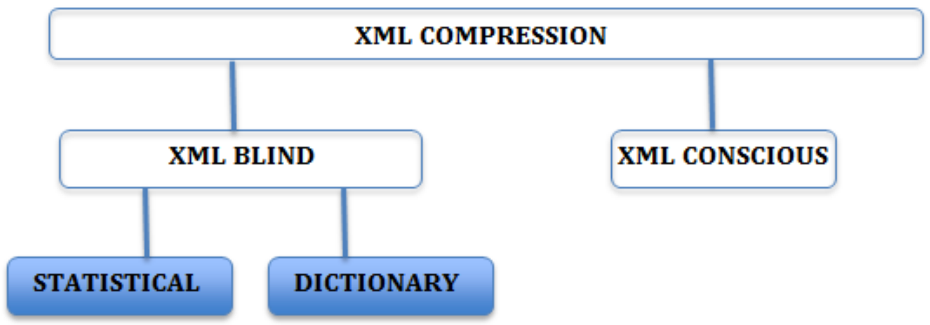


Figure 5.1: XML blind Compressors

5.3 XML Conscious Compressors

XML conscious compressors are the compressors that take the structure of XML into account during compression thereby achieving a better compression than XML blind compressors. These compressors can be categorised as either queryable or non-queryable. This classification is based on their query support. The *non-queryable* compressors do not support query evaluation, the whole document must be decompressed for any query to be evaluated. The *queryable* compressors on the other hand support query evaluation with partial decompression. Sometimes queries can be evaluated on the compressed data with no decompression.

Supporting efficient query evaluation comes at the expense of the compression ratio. A trade-off always exists between the compression ratio and support for query evaluation. Compressors that do not support query evaluation usually have a better compression ratio than those that do. These methods are discussed in Section 5.3.2 and 5.3.3 with examples of existing compressors for each category. Another classification of XML conscious compressors is based on their dependency on the XML schema. This classification is briefly discussed in Section 5.3.1 before the non queryable and queryable compressors because it overlaps. Some queryable and non queryable compressors are schema dependent thereby fall in the same class in this classification.

5.3.1 Schema Dependency Classification

XML compressors can use the schema of an XML document to achieve a slightly better compression time and ratio but some compressors do not depend on the schema. According to the schema dependency classification, compressors can either be schema dependent or schema independent. Theoretically, schema dependent compressors are better than the schema independent ones in that they achieve a better compression ratio and sometimes even a better compression time.

The schema dependent compressors are not widely used because schemas are sparingly employed on XML documents so there is no guarantee that the schema will be available hence the popularity of schema independent compressors. XML blind compressors are schema independent by default since they compress XML like general text. The XML conscious family has both schema dependent and schema independent compressors. As mentioned above this classification overlaps among the XML conscious compressors, that is some queryable methods are schema dependent whereas some are schema independent and the same applies to the non queryable. The following sections discuss queryable and non queryable compressors

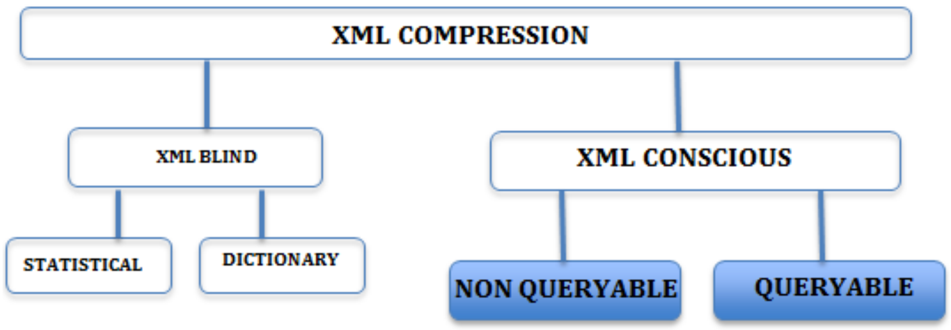


Figure 5.2: XML conscious compressors

5.3.2 Non-Queryable Compressors

Non-queryable compressors exploit the XML structure to achieve a better compression ratio than their queryable counterparts and as their name suggests they do not support query evaluation. With these methods, the whole document must be decompressed for a query to be evaluated. Most XML conscious non-queryable methods compress XML by separating the structure from the data. This is followed by putting data into different containers sometimes according to data types. The containers are then compressed using a general text compressor that is suitable for the data type in each container. This results in the highest compression ratios as suggested by the discussion below where each non queryable is presented.

XMill [Liefke and Suciu, 2000] was the first XML conscious non queryable compressor to be implemented. Consequently, this compressor set the basic ideas for compressing XML that were later adopted by other compressors. XMill compresses XML by separating the structure from data. The start tags are then assigned integers and the end tags are replaced by '/'. These are then put in a container that is compressed using gzip. The data is also grouped according to the relative path on the XML tree and data type. This process groups data that is semantically related to create homogenous containers. The homogenous containers are compressed using a semantic compressor achieving a high compression ratio.

Other compressors that are used with XMill besides gzip are PPM and BZIP2. XMill achieves a compression ratio that is almost twice that of gzip; however, this significant difference is not shown when working with smaller files that are about or less than 20KB [Nair, n.d.)]. This compressor is schema independent. XMill does not support queries. The whole document has to be decompressed for any query to be evaluated. It is therefore more useful in archiving data and data transmission; it requires lesser disk space and reduces the network bandwidth respectively.

Li presented another compressor that is similar to XMill; XComp [Li, 2003]. This compressor adopts the idea of separating structure and data from XMill. However, it adds another dimension in grouping data values. Instead of using the relative path and data type alone, XComp adds the level or depth as another option to be considered in this grouping. XComp therefore uses relative path, level and data type to group data values. As in XMill, the structure is compressed in a container and the grouped data values are also compressed in different containers. Each container has its own alphabet frequency where Huffman encoding is applied. During compression the size of each container is restricted so as not to exceed the size of the memory window. This is said to increase the efficiency of memory usage [Li, 2003]. Another distinguishing feature of this compressor is that it has a container for processing instructions and comments [Pena, 2013]. XComp is a schema independent compressor. In their work Hruska et al. claimed that XComp only showed a slight improvement from XMill [Hruska et al., 2010].

XMLPPM [Cheney, 2001] is a streaming non queryable compressor that uses Multiplexed Hierarchical PPM models [Cheney, 2001]. The compressor uses the SAX encoding scheme together with PPM [Cleary and Witten, 1984, Teahan and Cleary, 1996, Cleary and Teahan, 1997] which is a general text encoder. In parsing the XML document using SAX, the SAX events created are encoded using a bytecode representation called ESAX (Encoded SAX). The bytecodes are then compressed using one of the multiplexed PPM models according to their syntactic context.

XMLPPM is an adaptation of PPM where different structural parts (elements, attributes, characters) of an XML document are encoded using a unique model. The dependencies between elements and the data they enclose are retained by injecting the element symbol into the corresponding model. XMLPPM is claimed to generally have a better compression ratio than XMill but poor compression time since the PPM compression family is generally slower [Pena, 2013].

A variant compressor to XMLPPM called SCMPPM which combines Structure Context Modelling (SCM) [Adiego, Navarro and de la Fuente, 2003] with PPM instead ESAX was presented by Adiego et al. [Adiego, de la Puente and Navarro, 2004]. This compressor uses a separate model to compress text data under each distinct XML tag. As a result SCMPPM uses a bigger set of PPM models than XMLPPM. These authors also presented another SCM variant which uses the Huffman compression instead of the PPM models. With this variant a dictionary is created for each different tag. In some cases dictionaries have to be merged if they are equivalent because many dictionaries can be a challenge for storage space which then defeats the purpose of the compressor. The storage space flaw also applies to the SCMPPM.

AXECHOP is a grammar based compressor proposed by Leighton et al. [Leighton et al., 2005]. It is a schema independent non queryable compressor that separates structure from data like XMill. Unlike XMill, AXECHOP processes the structure using a byte tokenization scheme. This preserves the structure of the original document. An MPM algorithm [Kieffer et al., 2000] is then used to produce a context free grammar. This process is followed by encoding the grammar with an adaptive arithmetic encoder [Witten, Neal and

Cleary, 1987] before being written to a compressed file [Sark, 2009]. As with other compressors, the data part of the document is separated into containers according to the enclosing tag or attribute. Burrows Wheeler Transform (BWT) [Burrows and Wheeler, 1994] is applied to each container and the results are appended to the compressed file.

XBzip uses XBW transform [Ferragina et al., 2005] which is inspired by the BWT used in AXECHOP; however, this compressor uses PPM as the back end compressor [Ferragina et al., 2006]. Exalt [Toman et al.; 2004] like AXECHOP, is a grammar based XML compressor. Both compressors exploit the idea that XML can be defined by a context free grammar. Exalt uses a syntactical oriented approach to compress the XML documents. In this approach grammar based codes encoding technique is used to generate grammar incrementally which is then encoded using the adaptive arithmetic coding.

Another dictionary based compressor that adopts the XMill idea of separating structure from content data is XML Word Replacing Transform (XWRT) [Skibinski and Swacha, 2007]. As with XMill the data content is separated into containers but in this method the data is grouped according to element names instead of the whole paths from the root. A dictionary is created through a preliminary pass over the document and words that appear frequently are replaced by a reference to the dictionary. The dictionary is encoded using a byte-oriented prefix code. The encoded results are compressed using a suitable general text compressor: PPM, LZMA [Pavlov, n.d] and gzip.

XAUST [Hariharan and Shankar, 2005] is an online compressor that is schema dependent. This compressor converts the knowledge gathered from the DTD into a set of Deterministic Finite Automata (DFA) for each element. It uses the structure of an XML document from the DTD to accurately predict the expected symbols. Transitions of each automaton are labelled by element names.

The states may result in a single or multiple output transition wherein, no symbol encoding is required and each output transition is encoded using arithmetic encoder. Character data and attributes associated with each element are put in a container which is then compressed using the Arithmetic order-4 compressor [Cheney, 2006]

RNGzip [League and Eng, 2007] is another schema dependent compressor but it uses the RELAX NG schema instead of a DTD. With this compressor the receiver and the sender agree on a specific schema which then acts like an encryption and decryption key. The compressor builds a deterministic tree automaton from the schema. When given an XML document the compressor only needs to produce symbols since it would have already gathered some information from the schema. In producing these symbols upon encountering a choice point the transition taken is transmitted and when textual transition is encountered the textual data is transmitted. The streams generated from this process are compressed using a suitable compressor from gzip, BZIP LZMA and PPM which are all XML unconscious compressors. Other compressors that use schemas are the DTDPPM [Cheney, 2005] which also employs PPM like XMLPPM and Millau [Girardot and Sundaresan, 2000]. Millau is not purely schema dependent though; it only uses a schema if the schema is available otherwise compression can still be done without the schema.

5.3.2 Summary

It is necessary to discuss non-queryable compressors to understand how XML compression begun but these compressors are archival therefore not a solution for cases where compressed XML needs to be queried. They are only useful for storing large data in small spaces and very effective in data transmission over the internet. Non queryable compressors reduce the XML file size considerably thereby reducing the amount of disk space required. Small file sizes save bandwidth resulting in fast data transmission. However these compressors are not that useful for limited devices like smartphones where users do more than just storing and or transmitting data. Limited devices are widely used for data processing. Many users use their smartphones to manipulate data and this includes querying the data. As stated earlier non-queryable XML compressors do not support query evaluation. This posed a problem that resulted in the development of XML compressors that allow query evaluation over compressed data sometimes with partial decompression; the queryable compressors.

5.2.3 Queryable Compressors

Queryable XML compressors acquired their name from their ability to support query evaluation over compressed data sometimes with partial decompression. These compressors have two main goals; to achieve a reasonable compression ratio compared to non queryable compressors and to allow efficient query evaluation. The trade-off between compression ratio and query support is observed with these compressors. Compressors that have a high compression ratio normally support a narrow range of queries and most cases these methods require partial decompression more than those that have a lower compression ratio.

Queryable compressors include **XGRIND** [Tolani and Haritsa, 2000], **XPRESS** [Min, Park and Chung, 2003], **TREECHOP** [Leighton, Müldner and Diamond, 2005], **XQueC** [Arion et al., 2007], **XQzip** [Cheng and Ng, 2004], **SJXC** [Wei and Wei, 2012], **XPACK** [Rocco, Caverlee and Liu, 2005], **XCQ** [Ng et al., 2006], **XSeq** [Lin et al., 2005], **XCPaqs** [Wang et al., 2004], **ISX** [Wong, Lam and Shui, 2007], **QXT** [Skibinski and Swacha, 2007], **LZCS** [Adiego et.al., 2007], **XBzipINDEX**[Ferragina et al., 2006], **SXSI** [Arroyuelo et al., 2010] and **XSAQCT** [Müldner et al. 2009]. These compressors can be further classified as homomorphic and non-homomorphic. This classification is based on whether the compressor retains the tree structure of XML during compression or not.

The non homomorphic compressors compress data by separating the structure of XML from the content and as a result the compressed XML structure is not the same as the original structure. On the other hand, homomorphic compressors retain the tree structure such that the compressed structure can be parsed and queried like the original structure. With these compressors the structure is not separated from the content. This process results in low compression ratios but it preserves the opportunity to query the compressed XML without decompression. Therefore homomorphic compressors are not as popular as their non homomorphic counterparts.

5.2.3.1 Homomorphic Compressors

Homomorphic compressors include XGRIND, XPRESS and QXT. The first queryable homomorphic XML compressor, XGRIND, was presented about a decade ago [Tolani and Haritsa, 2002]. It is a schema dependent compression method that employs dictionary encoding and semi adaptive Huffman encoding. XGRIND compresses XML by creating a dictionary that stores all elements and attribute names shown in the document's DTD. The dictionary is used for encoding elements and attributes with dictionary encoder.

This compressor parses the input document twice. In the first parse the compressor collects statistics about the PCDATA in the input document to create coding models for the Huffman coders. The second parse is for tokenising all tag and attribute names using indexes to the corresponding entries in the dictionary. Numerical attribute values are binary encoded. All encoded tags, attributes and data are packed together with the dictionary as the output. The homomorphic nature of XGRIND allows query evaluation over compressed data but this is only possible for simple queries like exact match and prefix queries [Tolani and Haritsa, 2002].

Partial decompression is required to evaluate partial match and range queries. This compressor does not support other operation like aggregations, joins, nested queries and non-equality selections. In addition to the limited query support, XGRIND creates a time overhead during compression by scanning the XML file twice. Its compression time is said to be longer than that of XMill [Nair, n.d]. Zhang et al. supported these claims by stating that XGRIND compression time is twice XMill's [Zhang et al., 2008]. The compression ratio is also worse than that of XMill. Evidently query support compromises compression ratio.

Min et al. presented XPRESS as an improvement on XGRIND [Min et al., 2003]. XPRESS has a better compression ratio and supports more queries [Arion et al., 2010] on the compressed data than XGRIND and this reduces partial decompression overheads [Nair, n.d]. However, evaluating textual range queries in XPRESS requires partial decompression just like in XGRIND. XPRESS has a better query response time [Nair, n.d]. Like XGRIND, this compressor is also homomorphic and it scans the XML file twice for compression. XPRESS does not use Huffman encoding instead it adopts the reverse arithmetic encoding. In XPRESS data is compressed by mapping the entire XML hierarchy over the real interval (0.0, 1.0). As with XGRIND, XPRESS does not support complex operations such as structural joins [Wei and Wei, 2012].

Another homomorphic compressor is QXT [Sibinski and Swacha, 2007]. This compressor is an extension of XWRT. It is extended to allow query processing. Unlike the XBWT which compresses XML according to element names, this compressor uses paths to separate data into containers. Like other homomorphic compressors it scans the XML file twice. The first scan is to gather statistics or frequencies of each item. The second scan is the actual transformation stage where data is tokenized and put into containers according to their paths from the root.

The containers are memory buffered until the reach a specific threshold after which they are compressed using a back end compressor. The containers are compressed into 32KB blocks to allow easy partial decompression. QXT processes queries by reading the dictionary from the compressed file first. This is followed by determining the containers that may contain the data that matches the input query. These containers are decompressed and the transformed data from the containers is searched using a transformed pattern. Only the result set is decoded to the original XML format.

5.2.3.1 Summary Homomorphic Compressors

From the above discussions, it is clear that homomorphic compressors scan the input document twice. They all scan XML first to collect statistics about the frequency of items to build a dictionary. The second scan is where the actual compression takes place by tokenizing the data for XGrind and QXT. From the above discussion, QXT is the only homomorphic compressor that encodes paths instead of considering individual elements. These compressors allow efficient query processing for the supported queries; however, they do not support all operations.

5.2.3.2 Non Homomorphic Compressors

Non homomorphism is adopted by many queryable compressors because it provides more flexibility. All other queryable compressors mentioned in this chapter besides XGRIND, XPRESS and QXT are non homomorphic. These methods do not retain the structure of the original document but they allow query processing mostly with partial decompression.

5.2.3.2.1 Path Compressors

XCPaqs [Wang et al., 2004] is an example of a compressor that separates structure from content in compressing XML. Like XGRIND, XPRESS and QXT, XCPaqs scans the XML file twice. In its initial scan statistics about tags, path and path types are collected. The structure is compressed separately from the content. To compress the structure tags are compressed first using a Huffman encoder based on the statistics collected during the scan. This is followed by compressing paths in the same manner.

In compressing the content, path types that are gathered in the initial scan are considered resulting in data encoded according to the type inferred by the path type. The enumerated data is therefore compressed using a dictionary encoder whereas the string type is compressed using a suffix compressor. Long text on the other hand is compressed using the Burrows Wheeler Transform. The connection between structure and content is maintained by the path order in the original document. This relational connection is the one that is used in the final stage to form a *2-ary* structure. XCPaqs supports XQuery queries however an input query has to be translated into a corresponding XCPaqs code.

XSAQCT [Müldner et al., 2009] merges similar paths to localize repetition to encode data. Like XCPaqs, this compressor uses paths in its compression process. However, in this compressor the structural similarity of paths is significant, not the path type. Each node is annoted with a set of integers depicting the nodes along that path. All this is done within a single SAX traversal over the XML document. Thereafter, the annotations are written to containers that are compressed using some back end compressor. XSAQCT adopts gzip as a back end compressor. It also uses Bzip2 and PAQ8 [Müldner et al., 2009]. Containers with the required data are decompressed during query evaluation. XSAQCT supports simple exact match queries.

5.2.3.2.2 Sub Tree Compressors

Other non homomorphic compressors consider sub trees instead of paths. XQzip [Cheng and Ng, 2004] is one of the compressors that use sub trees during compression. It employs a Structured Index Tree (SIT) and a buffer pool to compress XML. SIT merges sub trees that have equivalent paths to remove redundancy and improve query performance. Merging the sub trees results in a sequence of block partitions that are then compressed using gzip. This compressor has a compression ratio close to that of XMill [Cheng and Ng, 2004]. During query processing, only the required blocks are decompressed. Using the query input, the query processor determines the minimum number of block necessary to evaluate it.

Bigger block sizes achieve a better compression ratio but suffer a decompression overhead and that greatly affects query performance. Smaller blocks result in redundancies between blocks compromising the compression ratio. With that, the critical part of this compressor is determining the block size. To further alleviate the decompression overhead problem, an algorithm called Least Recently Used (LRU) is applied to maintain a buffer pool of the decompressed blocks thus avoiding repeated decompression.

XQzip supports extended XPATH queries including the union and grouping operators [Arion et al., 2010]. It supports a wider scope of queries than XGRIND and XPRESS [Hruska et al., 2010], however, it does not support joins and order based predicates [Zhang et al., 2008, Ng et al., 2006]. Query evaluation with this compressor is very fast compared to other methods but the efficiency is significantly reduced when dealing with queries with descendant Ancestor-Descendant edges.

LZCS is a non homomorphic queryable compressor that uses the Lempel-Ziv approach. Lempel-Ziv approach exploits repeated information (redundant sub trees). A repeating sub tree is replaced by a backward reference to its first occurrence. LZCS can be said to merge sub trees like XQzip because replacing a sub tree with a backward reference of its first occurrence is in a way merging similar sub trees. This process results in a compressed document that is easy to display, access at random and navigate [Adiego et al., 2004]. The compressed document can further be compressed by a semi static word based Huffman method [Moffat, 1989] or PPM schemes [Cleary and Witten, 1984]. The latter does not keep the navigation properties of LZCS [Pena, 2013]. LZCS does not perform well on generated semi structured data but is better on highly structured documents [Adiego et al., 2007].

Sub trees are also used in XMLZIP. However, unlike XQzip and LZCS it does not merge them but rather divides them into different containers as they are. Compressing XML with this compressor entails dividing a DOM tree at depth **d** specified by the user into multiple components [Cheney, 2001; Pena, 2013]. It has been established that increasing **d** decreases the compression ratio because the redundancies in separated trees cannot be exploited to improve compression [Cheney, 2001]. The component that contains all nodes at a given depth **d** is referred to as the root component. This component is modified by adding references to the sub trees it contains. The multiple components generated by the above process are then compressed using gzip. It is said to have a poor compression ratio compared to the generic gzip [Ng et al, 2006; Cheney, 2001]. Its only advantage over gzip is that it allows limited random access to partially decompressed XML documents because, with gzip, full decompression is required.

5.2.3.2.3 Auxiliary Structures Compressors

Some compressors like XBzipINDEX, ISX and SXSI use auxiliary structures like indexes to improve the efficiency of their query evaluation process. Some of these compressors adopt this so well that no decompression is required to evaluate queries. However, there are challenges that come with having auxiliary structures in that the compression ratio is affected because there is more information to store so this has to be carefully considered. These methods are discussed below.

XBzipINDEX [Ferragina et al., 2006] is a compression tool that adopts the XBW transform [Ferragina et al., 2005] to compress XML just like the XBzip that is discussed earlier in this chapter; however, XBzipINDEX is searchable because of the index it incorporates. XWB portrays a succinct tree using two arrays. One array stores tree labels arranged in appropriate order while the other is a binary encoding array of the structure of the tree. The additional data structures are required to support selection and rank operations over the two XBW arrays. To achieve this, the arrays are handled as strings to employ a string index tool; the FM-index [Ferragina and Manzini, 2005]. This index supports searching the tree. XBzipINDEX is said to have a compression ratio that is 35% better than that of XGRIND, XPRESS and XQzip [Ferragina et al., 2006]. This makes it one of the compressors that have successfully adopted the idea of employing auxiliary structures without compromising the compression ratio. It is also said to be the first compressor to employ an index in compression [Ferragina et al., 2006].

Another auxiliary structure compressor, XQuec [Arion et al., 2007], compresses data by encoding it with a simple binary encoding. It splits the XML document into three data structures; structure tree, containers and the structure summary or DataGuide. The structure tree is a set of ID sequences that identify each root-node path in the tree. All the distinct paths in the document are stored in the structure summary. Data and tag values are also split according to their root to leaf path and stored in containers. Clearly this compressor relies heavily on path information in compressing data. XQuec, like XSAQCT, uses path information but unlike XSAQCT it uses the information to split data into containers not to merge similar paths. XQuec preserves relationships by storing distinct paths and IDs in the DataGuide and structure tree respectively. Containers in XQuec are compressed using ALM [Antoshenkov, 1997] or the classical Huffman algorithm.

To choose a suitable back end compressor a cost model is devised by exploiting the query workload information. ALM is used to support range queries and inequality predicates because it preserves order. The Huffman algorithm on the other hand is used in cases where prefix wildcards need to be supported but not inequality. As seen from the above discussion, the focus of this compressor is more to support a wide spectrum of queries efficiently rather than achieve a better compression ratio.

XQuec generates a lot of pointers that enable it to support the evaluation of a wide range of XQuery queries in its compressed state [Wei and Wei, 2012]. It also has a better query performance than XPRESS and XGRIND [Zhang et al., 2008]. However, it comes at the expense of the compression ratio. As aforementioned, XQuec has many auxiliary structures like the structure tree and DataGuide which enables it to support many queries but these structures may sustain a storage overhead compromising the compression ratio which defeats the purpose of it being a compressor.

Wong et al. presented a compact storage engine for XML, ISX [Wong et al., 2007] that also utilizes auxiliary structures. This storage engine is classified as a compressor because of the way XML documents are stored in it. ISX has three layers; the topology layer, internal layer and the leaf node layer. Each layer stores specific components of the XML document. The topology layer employs balanced parenthesis encoding proposed by Katajainen and Mäkinen [Katajainen and Mäkinen, 1990] to store the tree structure of the XML document. The internal layer stores elements, tags and signatures to the text data. The actual data is compressed by a common compressor like gzip and gets stored in the leaf node layer. The topology layer references this data for easy access. Auxiliary data structures are used over the topology layer to allow direct node navigation thus supporting query evaluation over the compressed data. Other operations that ISX supports are update functions like insertions and deletions. This makes it competitive in query support but it still suffers from a low compression ratio.

SXSI is another compression tool that uses an index [Wong et al., 2007]. This compressor uses an FM-index [Arroyuelo et al., 2010]. XML documents are treated as a set of ordered strings and a labelled tree defined by a hierarchy of tags. This approach separates the structure of the XML tree from the text content. Each node in the structure is assigned a *global identifier* and each text content is assigned a *text identifier*. Text data is concatenated and represented using the FM-index. SXSI uses this index to gather information that allows it to only visit relevant nodes during query evaluation, thereby reducing processing times. SXSI supports a wider range of XPATH queries than XBzipINDEX [Arroyuelo et al., 2010].

5.2.3.2.4 Online Compressors

Other non hormomorphic queryable compressors are online compressors. These compressors are especially applicable to easy data transfer over the network because they lower the bandwidth required to transfer data. Two online compressors are discussed in this work; TREECHOP and XSeq. Although these compressors are both online compressors they use different strategies to compress data as discussed below.

TREECHOP [Leighton et al., 2005] is non homomorphic queryable online compressor that compresses data by assigning a codeword for a path to each non leaf node. Compression in both XSAQCT and TREECHOP is based on paths; XSAQCT merges paths instead of encoding them individually like

TREECHOP. With TREECHOP one codeword is assigned to each absolute path such that nodes with the same absolute path have the same codeword. A codeword for each node is prefixed by its parent's codeword. New tree nodes are written to the compression stream in depth first order as the SAX tokens are returned. This depth first order enables the decompressor to regenerate the original document [Sakr, 2009]. It avoids building an in memory representation of the entire XML tree to preserve resources. TREECHOP supports exact match and range queries [Leighton et al., 2005]. Unlike other methods it evaluates these queries over the compression stream instead of decompressing relevant containers because it does not split data into containers. Partial decompression is required for validation when processing range queries [Pena, 2013].

XSeq [Lin et al., 2006] adopts a grammar based text string compression algorithm, Sequitur [Craig et al., 1997] instead of the codewords used in TREECHOP. This online compressor uses Sequitur to generate a context free grammar that uniquely represents the input string. In this compressor tokens are separated into a set of containers each of which is compressed by Sequitur. XSeq also adopts the use of auxiliary structures like XBzipINDEX, ISX and SXSI but it uses them in a slightly different way. It requires these auxiliary data structures or indexes to be loaded in memory before processing the rule contents [Sark, 2009]. It primarily has two indexes; the structural index and the header index. The indexes allow data values to be quickly located and provide pointers to the entrance of each container respectively. They enhance the compressor such that no decompression is required during query evaluation.

5.2.3.2.5 Other Compressors

This section discuses a few other compressors that do not fall in the above classifications. They include XCQ which is the only queryable compressor that uses a schema, SJXC which is a more advanced compressor that supports more queries than all other compressors and XPACK that employs a heavy use of containers in its compression process.

Most non hormomorphic queryable compressors employ containers during compression but this idea is overly used in XPACK [Rocco, Caverlee and Liu, 2005]. XPACK adopts a container oriented document structure that is created and modified by a set of unary operators [Rocco et al., 2005]. These operators include *PagePack(), PathPack(), NamePack(), URLPack(), AttributePack()*, *ContentPack()*. The operators are responsible for document container, path structure, node tag name, document URL, attribute and content encoding respectively. The main aim of XPACK is redundancy elimination so its operators support flexible redundancy reduction. The *PagePack* operator is different from other operators since it is the only one that operates on the original XML document whereas other operators operate on containerized documents. This operator generates a compact representation of the document structure augmented by a set of content containers. These containers are then replaced, transformed or augmented by the relevant operators. XPACK has a good compression ratio and strong query support for compressed XML documents [Rocco, Caverlee and Liu, 2005]. It is claimed to reduce the storage requirement of XML by up to 20% over XGRIND and XPRESS [Rocco, Caverlee and Liu, 2005].

In recent years compressors that are more advanced have been presented. These compressors use complex strategies to compress data and also support a wider range of queries. SJXC [Wei and Wei, 2012] is one of the latest compressors. It uses a region encoding method called begin-end to encode nodes in an XML document. The document is parsed using SAX and tags and attributes are separated from the data. Data is then separated into different containers according to their paths. These containers are encoded using incremental coding or dictionary encoding depending on the type of data they contain. Numerical data is encoded using incremental encoding while textual data is encoded using the latter.

Given an input query in SJXC, a query parser changes the query into a codeword expression. This expression is then matched against data in the containers. If there is a match the desired data is decompressed. SJXC adopts the TwigStack algorithm [Zhou, Xie and Meng, 2007] to evaluate twig queries thus caters for complex operations like structural joins.

As with non queryable compressors some queryable compressors are schema dependent therefore require the availability of a schema during the compression process. Ng et al. presented a schema dependent queryable compressor, XCQ [Ng et al., 2006]. As with other schema dependent compressors, XCQ exploits information provided by the DTD to achieve a better compression ratio. It adopts a technique called DTD and SAX Event Stream Parsing. In this technique, documents are partitioned according to their path and stored in blocks that are indexed using a block statistics signature scheme [Sakr, 2009]. The blocks and the structure streams are compressed using *gzip*.

In the same way as XQzip, XQC compression ratio and query performance are inversely affected [Sakr, 2009] because of the decompression overhead during query evaluation. With this compressor queries are evaluated with partial decompression. Only the necessary blocks are decompressed. XCQ only supports a subset of XPATH queries including selection and aggregation [Zhang et al., 2008, Sakr, 2009]

5.2.3.2 Summary Non Homomorphic Queryable

The main focus for Non homomorphic queryable compressors is to find the best possible way to achieve a better compression ratio and maintain some support for query evaluation over the compressed data sometimes with partial decompression. Most of them employ the use of containers where data is separated into containers using different strategies. Data can be separated using absolute paths, node names or sub trees. Sometimes similar paths and sub trees are merged to eliminate redundancy. In some cases to further reduce the size, some form of encoding is used. This ranges from simple integers to special compression encoding like binary encoding. Containers are then compressed using a back end compressor which is normally an XML blind compressor mostly gzip.

In a case where a compressor supports query evaluation with partial decompression, relevant containers are decompressed during query evaluation. As discussed above, some compressors use auxiliary structures to avoid the need to decompress data during query processing. From the above discussion it is clear that some methods choose to compromise compression ratio to allow a wider query support. Though the compression ratio is compromised using these compressors is still better than processing the original XML document because the size is still somewhat reduced.

5.2.3 Summary Queryable compressors

All the discussed queryable methods allow query evaluation on the compressed data but sometimes partial decompression is required. These methods mostly compromise compression ratio for query support. Some queryable compressors retain the structure of XML during compression therefore are referred to as hormomophic compressors while others do not. Compressors that are non hormomophic usually result in a better compression ratio.

Note that all these compressors were developed in light of reducing space consumption in a resource rich environment but not specifically for resource constrained devices. Therefore they do not meet the needs of a resource constrained device especially in terms of query processing.

5.4 Chapter Summary

The verbose nature of XML resulted in a lot of research into XML compression. It has been established that XML can be compressed with general text compressors because it is a text file. However, using general text compressors result in low compression ratios. These compressors are referred to as XML blind compressors because they are unaware of the structure of XML. For this reason they cannot exploit the repetitive structure of XML to achieve better compression.

This led to XML conscious compressors that exploit the repetitive structure of XML thereby achieving considerably high compression ratios. The XML conscious compressors are further classified into queryable and non queryabe compressors based on their ability to support queries. Non queryable methods have higher compression ratios than queryable methods. It has been observed that there is a tradeoff between compression ratio and query support. Compressors with a high compression ratio either support a few queries or none at all. For the queryable methods, some compress XML by retaining its structure to allow support more queries. These methods are classified as homomorphic compressors. The non homormophic methods have a better compression ratio than their hormomophic counterparts. Research in queryable compressors has advanced but there has not been a method that has been developed specifically for resource restrained devices.

CHAPTER 6

MOTIVATION AND PROBLEM FORMULATION

6.1 Introduction

This chapter discusses the motivation of this research by outlining the importance of XML highlighting its verbosity problem. It also encapsulates XML compression. In addition, the subject of smartphones is presented since the main aim of this research is to investigate how labelling schemes can be used alongside existing compression methods to achieve a competitive compression ratio that to some extent allows users to store and manipulate the otherwise verbose XML in resource-restricted devices. Section 6.2 gives an overview of XML while Section 6.3 discusses smartphones. The use of wireless connections as a solution to some of the problems that come along with using smartphones is discussed in Section 6.3.1 and Section 6.4 considers XML compression background showing some the limitations of the existing compressors. The chapter concludes by briefly presenting labelling schemes leading to the problem definition and hypothesis of this research.

6.2 XML overview

The explosive growth of the use of XML over the last decade has led to a lot of research on how to best store and manipulate it. XML has been described as a de facto standard for storage and exchange of data over the web [Sakr, 2009; Ng et al., 2006; Lu and Cheng, 2004; Nicola and Van der Linden, 2005; Weiner, Mathis and Härder, 2008; Su-Cheng et al., 2009; Haw and Lee, 2007; Zhou et al., 2009; Wang et al., 2009; Mlynkova and Necasky, 2009; Grimsmo and Bjørklund, 2010; O'Connor and Roantree, 2010; Zhang and Özsu, 2010; Byun and Park, 2010, Xin, He and Cao, 2010].

Though it is verbose, XML cannot be phased out because of its capability to represent data in a simple way that can be understood by both humans and machines [Kirk et al., 2005]. XML is platform independent [Nicola and Linden, 2005] providing portability [Gulhaneand Ali, 2012; Morgan, 2007] which is a desirable feature with the diverse users available. Among other advantages XML has a fairly simple syntax and extensible vocabulary with user defined tags. These tags carry semantics about data that other languages cannot reflect thus it said to be a self-describing language. The self-describing nature of XML results in large documents [Arion et al., 2007; Maneth et al., 2008; Zhou et al., 2010; Ng et al., 2006; Arion et al., 2004] that may otherwise prove to be inaccessible for resource-restrained devices such as smartphones.

6.3 Smartphones

Equally there has been an explosive growth of smartphone use which has been driven by consumer demand [Satyanarayanan et al., 2013]. These devices have pervaded our lives [Boulos et al., 2011 Verbelen et al., 2013] in that users use them for almost everything that used to be achieved by using a desktop from checking emails to banking [Whang et al., 2009 Portokalidis et al., 2010]. Although users perform almost every task with smartphones there are still challenges that come about with this because these devices are resource-restrained. Over the years their capabilities have been increased but they still cannot compete with their resource-rich counterparts [Verbelen et al., 2013]. The challenges that come with using a smartphone include limited memory, finite energy and a slow processor [Boulos et al., 2011]. The storage problem has wireless connection being used as a solutions but it has some limitations as discussed in the next section.

6.3.1 Wireless connections as a solution

In trying to solve the limited memory problem to leverage the ubiquity of these devices, researchers proposed methods that rely on wireless connections (WI-FI) to keep most of the information on the server side and only keep the necessary information on the smartphone at a given time. Though it may have ubiquitous coverage, WI-FI has higher energy cost [Mohan, Padmanabhan and Ramjee, 2008] making it inadequate for many applications [Rahmati and Zhong, 2007]. The battery technology is one of the greatest obstacles for smartphones and their growth and the technology trends indicate that the problem is here to stay [Cuervo et al., 2010] therefore high energy processes must be avoided as much as possible.

Besides, their adverse effects on the battery life of a smartphone, wireless connections also comes with other challenges like security and the possibility of compromised data integrity. Apart from these problems wireless connections are not always available [Riva and Kangasharju, 2008, Satyanarayanan, 1996, Lindholm and Kangasharju, 2008]. When they are not, the user cannot have access to the data that resides in the server. Wireless connections may not available in some developing countries, in most aeroplanes and other hostile environments [Satyanarayanan et al., 2013] therefore cannot be viewed as a sustainable storage solution.

Although wireless connections can be used as a solution they come at a high cost. A more cost effective solution would be the one that increases physical proximity by making the XML document to some extent be small enough to fit in a smartphone yet accessible for query evaluation. This can be achieved by compression. Many compression methods have been proposed to reduce the size of the XML tree but none of them were developed in light of the smartphone use and the challenges that come with it.

This research is aimed at reducing the dependency on wireless connections by developing a compression method that has a competitive compression ratio and allows query evaluation of a wide range of XPATH operators without taxing the limited memory of a smartphone.

6.4 XML compression

The verbose nature of XML resulted in much work devoted to XML compression. Compression of XML has been achieved through XML blind compressors that compress XML like a text file, non-queryable and queryable XML compressors. XML blind compressors are unaware of the structure of XML so they result in poor compression ratios because they do not exploit the redundancy in the XML data structure. Non-queryable XML compressors, as shown in the Chapter 5, achieve better compression ratios than both the XML blind and queryable compressors but the resulting compressed file is inaccessible. These methods can only be a solution for storage space where file access is not required. The whole file needs to be decompressed to be manipulated so the non-queryable XML compression methods are not a feasible solution for smartphone limitations.

Queryable XML compression methods on the other hand allow parts of the XML file to be accessed at a time through partial decompression. Some of them allow data access over the compressed file. However, none of these methods were developed for smartphones. As discussed in the previous chapter these compression methods allow data access by compromising the compression ratio. Currently the best compression ratio of a queryable XML compressor is that of XSAQCT with a compression ratio of 0.8 [Al-Hamadani, 2011] but it supports the exact match queries only.

The way the data is stored influences the way it is retrieved. According to Pena more novel approach has been to combine compression and indexing such that an index is built on top of the compressed file resulting in a self-indexed compressor [Pena, 2013]. These compressors promise fast query execution because they employ indexes but this comes with high space requirement and that defeats the purpose of a compressor.

Compression changes the structure of the XML file and sometimes it requires many lines of code to extract data from such files. As already shown in Chapter 4 among other problems smartphones also have energy and processor limitations therefore executing complex lines of code may prove to be inefficient. It is necessary that a queryable compressor be developed specially for smartphones. The query processor has to be developed such that it can run on a smartphone considering the resource challenges to enable query processing efficiency. None of the existing compressors were developed such that their query processor can run on a smartphone. Evidently, there is still need to develop a compressor specially tailored for smartphones that can allow users to access their XML data at any given time at a minimal cost in terms of storage, power and time.

6.5 Labelling schemes

As discussed in Chapter 3 on query optimization, labelling schemes can reduce the size of an XML tree substantially yet preserving relationships among nodes in the XML tree [Weigel et al., 2005, O'Connor and Roantree 2010]. These relationships are core to XML query processing. Although labelling schemes reduce the size of an XML tree their primary purpose is to allow easy and fast access of data than reducing memory consumption. This research investigates how ideas from labelling can be combined with compression techniques to achieve a competitive compression ratio and allow easy and fast access to data.

6.6 Hypothesis

In light of the above discussion this research aims to test the following hypothesis:

"Combining ideas from XML labelling schemes and compression techniques can prune an XML tree enough to achieve a competitive compression ratio without the use of a schema that will to some extent allow users to store and query the otherwise verbose XML files in their resource-restrained smartphones efficiently."

The above hypothesis is tested through a practical implementation that is evaluated against real data that is usually used for XML benchmarking. The evaluation covers the compression ratio, compression time and query evaluation efficiency in terms of accuracy and speed (Query response time).

6.6.1 Objectives

Following the practical implementation this research addresses the following research questions:

1. Can labelling be combined with compression to achieve a competitive compression ratio?
2. Is it possible to achieve a competitive compression ratio without exploiting the XML schema?
3. What type of XPATH queries can be evaluated over the resulting compressed file on a smartphone?

This research hopes to have three contributions; the compressor, decompressor and query processor. Only the query processor runs on a smartphone, compressor and decompressor run on a resource- rich environment. These are briefly discussed below:

Compressor:	A schema independent XML compressor that combines labelling with compression achieving a competitive compression ratio that allows comparatively large XML files to be stored and manipulated in a smartphone.
Query processor:	A novel approach to XML query processing in smartphones that supports a wide range of XPATH operators. It evaluates queries over compressed data by retrieving relevant containers to get the desired data as specified by the query.

Decompressor:	The decompressor accepts the compressed files created by the compressor to produce the original XML document.

6.7 Chapter Summary

The use of smartphones is on the rise. Some schools of thought suggest that they will take over in the near future [Hu et al., 2010; Boulos et al., 2011; Kim et al., 2012]. However, these devices are resource-restrained and it will take years to overcome some of the challenges that come with their use especially the battery limitation. Smartphones are meant to be light and compact therefore some things like fans that are needed to dissipate heat cannot be incorporated into them. Not incorporating a fan means the processor speed has to be kept minimal otherwise the devices will heat up and be damaged.

Applications that are meant to run on a smartphone should be lightweight in terms of size and complexity of process to be executed. XML defies both properties in that it is verbose. Although it is verbose, XML has other desirable properties that have seen the increase in its use in the past decade therefore there is need to develop a novel approach by which XML can be easily stored and processed on a smartphone.

This is to some extent achieved by SIQXC which has a light weight query processor that allows efficient XML query processing on a smartphone. Obviously the capabilities of smartphones will increase with time but even then SIQXC will continue to be valid. As smartphones become powerful SIQXC can still be applied to compress XML such that more data can be carried and processed. The verbosity problem of XML will never change thereby making this compressor valid.

CHAPTER 7

SCHEMA INDEPENDENT QUERYABLE XML COMPRESSOR

7.1 Introduction

This chapter presents a Schema Independent Queryable XML Compressor SIQXC. This compressor was developed to alleviate problems that are associated with the verbose nature of XML so that it can be stored and manipulated in resource limited devices. As aforementioned, though XML is verbose its usage has increased and continues to [Pena, 2013, Al-Hamadani, 2011]. The use of limited devices like smartphones also has increased [Boulos et al., 2011, Satyanarayanan et al., 2013, Verbelen et al., 2013]. These devices are being used for different activities from personal to corporate and or business. Their resource limitations however make it almost impossible to handle XML without the aid of the Internet through the use of wireless connections. SIQXC therefore proposes a novel approach to compress XML such that large XML file would to some extent be stored and manipulated in these devices without the need for the Internet. Not only will this address the memory consumption problem but also other problems that come along with using the Internet like high energy consumption.

7.2 System overview

SIQXC consists of three main components; the compressor, decompressor and query processor. As already discussed in the previous chapter the main aim of this research is to investigate to what extent XML can be compressed such that users can have access to their data using a resource restrained device like a smartphone without the need to connect to the Internet. The compressor makes data small to some extent to fit in smartphone while the query processor gives accessibility to the compressed data again without the need for the Internet.

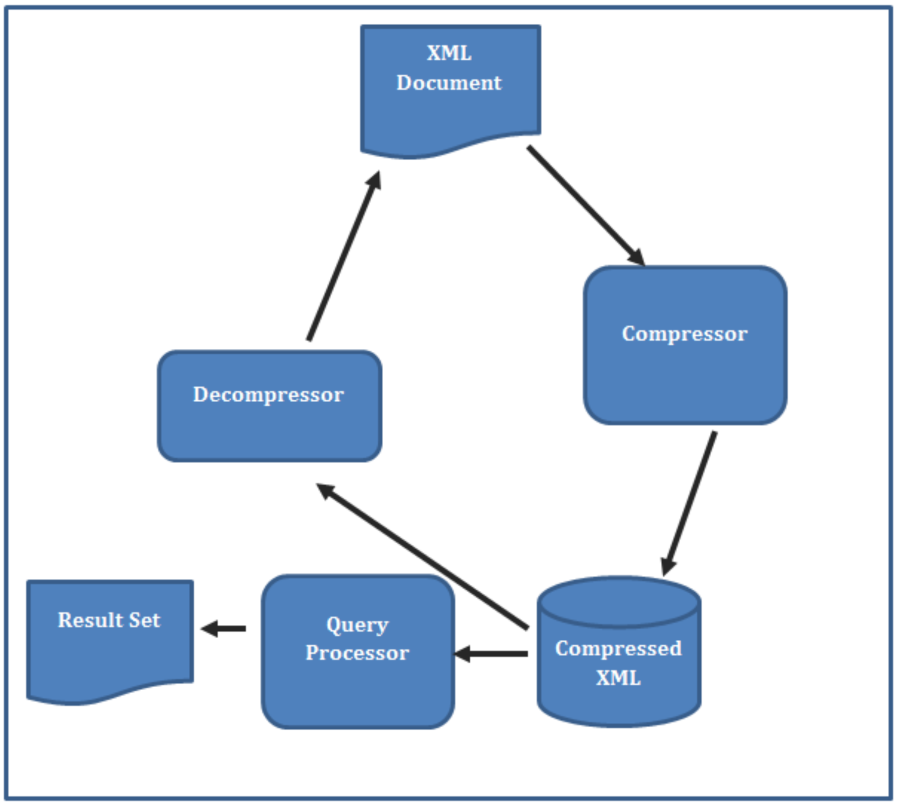


Figure 7.1: System overview design for SIQXC

SIQXC is designed such that these two components run in different environments. The compression stage runs in a resource rich environment while the query processor is developed for and runs in a resource limited device. This system also has a decompressor that decodes a SIQXC compressed XML file to the original XML document. Decompression can be necessary in cases where a user transfers a file to a different location and also for updates since data keeps evolving. However, the decompressor in this work was designed only for completeness since all other compressors have a decompressor. It is not useful in this work because the query processor does not support updates and there are other means of transferring files.

SIQXC accepts an XML document and compresses to produce a file of a smaller size. The compressed file can either be an input to the query processor or the decompressor. The decompressor decodes the compressed file back to the original XML. The file can be loaded into the query processor and accessed at any time by the user without the need for the wireless connections. Figure 7.1 above shows the system overview of SIQXC.

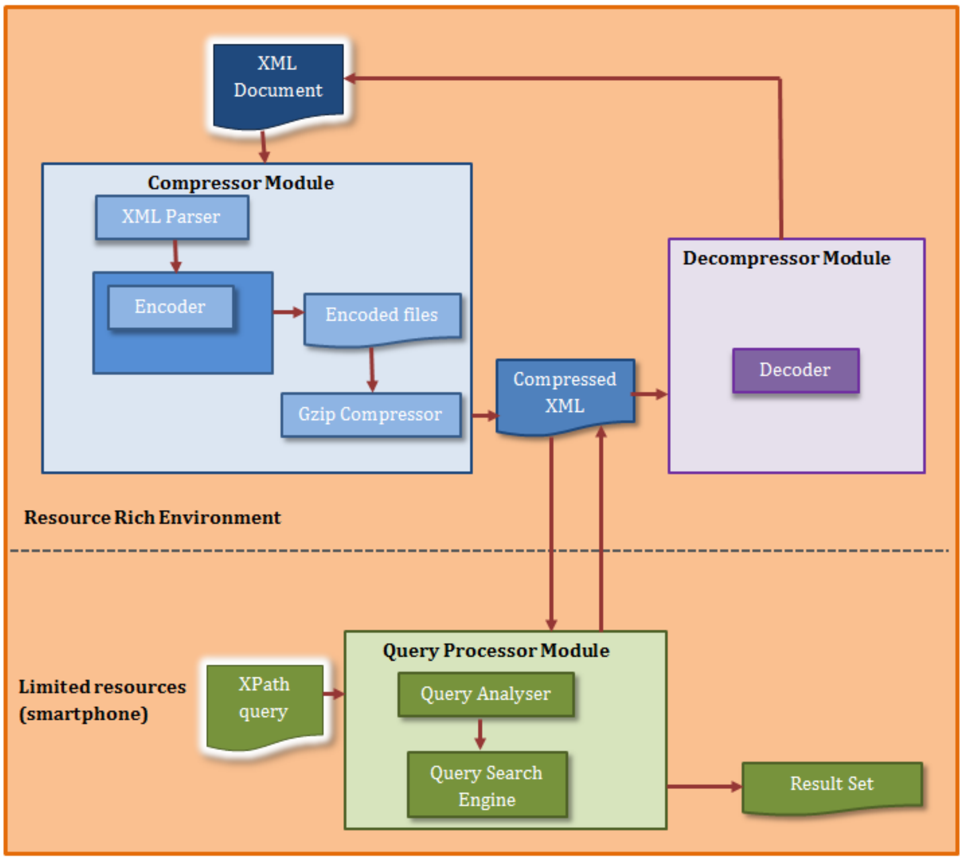


Figure 7.2: Complete SIQXC design

As stated above SIQXC has three main components; the compressor, query processor and a decompressor. The detailed design of this system is shown in Figure 7.2 above.

7.2.1 The Compressor

This compressor is schema independent to increase usability since the use of a schema bring about a lot of restrictions [Meng et al., 2003; Bex, Neven and Vansummeren, 2000] that reduce the flexibility of XML and lowers the number of users by excluding novice users. The schema could not be accessible to these users even if it was available [Al-Hamadani et al., 2013]. Not using a schema also saves space [Al-Hamadani et al., 2013] which is the main reason behind compression. It compresses XML by assigning the same code to siblings on an XML tree except for the siblings in level 2. Each node is labelled using a novel simple 2-tuple integer encoding system (see the next chapter).

The compressor also uses a node clustering technique to reduce redundancy and compresses the XML file further. The Node clustering technique was employed in Node Clustering Indexing Method (NCIM) to compress repetitive sub structures [Liao, Hsu and Chen, 2010]. NCIM combines indexing with compression and like any compressor that has auxiliary structures, needs more storage space even though it has a high compression ratio. In this node clustering process, nodes within the same level with the same node name are clustered together into a container with their data. This grouping reduces the size of the XML file significantly. The different containers are then compressed using DEFLATE [Deutsch, 1996, Weimin et al., 2008] based compressor gzip [Deutsch, 1996]. These techniques are discussed more fully in the next chapter.

7.2.2 The Query Processor

The query processor accepts XPATH queries. These queries are analysed by the query analyser to determine their type. Then queries are applied to the compressed XML producing a result set. The result set can be empty in cases where there is no match. Different methods are invoked to evaluate different types of queries. This query processor supports a wide range of XPATH operators. A detailed discussion of the query processor follows in Chapter 9.

7.2.3 The Decompressor

The decompressor decodes the compressed XML to the original XML document by unzipping, ungrouping and reassembling the different nodes to their respective levels and sub trees. This finishes by recreating the original XML file.

7.3 Chapter Summary

This chapter introduces SIQXC by giving an overview of its components and showing a detailed design diagram of the system. SIQXC has three main components; the compressor, decompressor and query processor. The compressor compresses XML by encoding, grouping and gzipping it. This is discussed in Chapter 8. This system supports a wide range of XPATH operators through the query processor. The original XML file can be obtained from the compressed file by decoding it with decompressor. The next chapter discusses the compressor and decompressor with examples to illustrate the procedures involved in each component.

CHAPTER 8

THE COMPRESSOR AND DECOMPRESSOR

8.1 Introduction

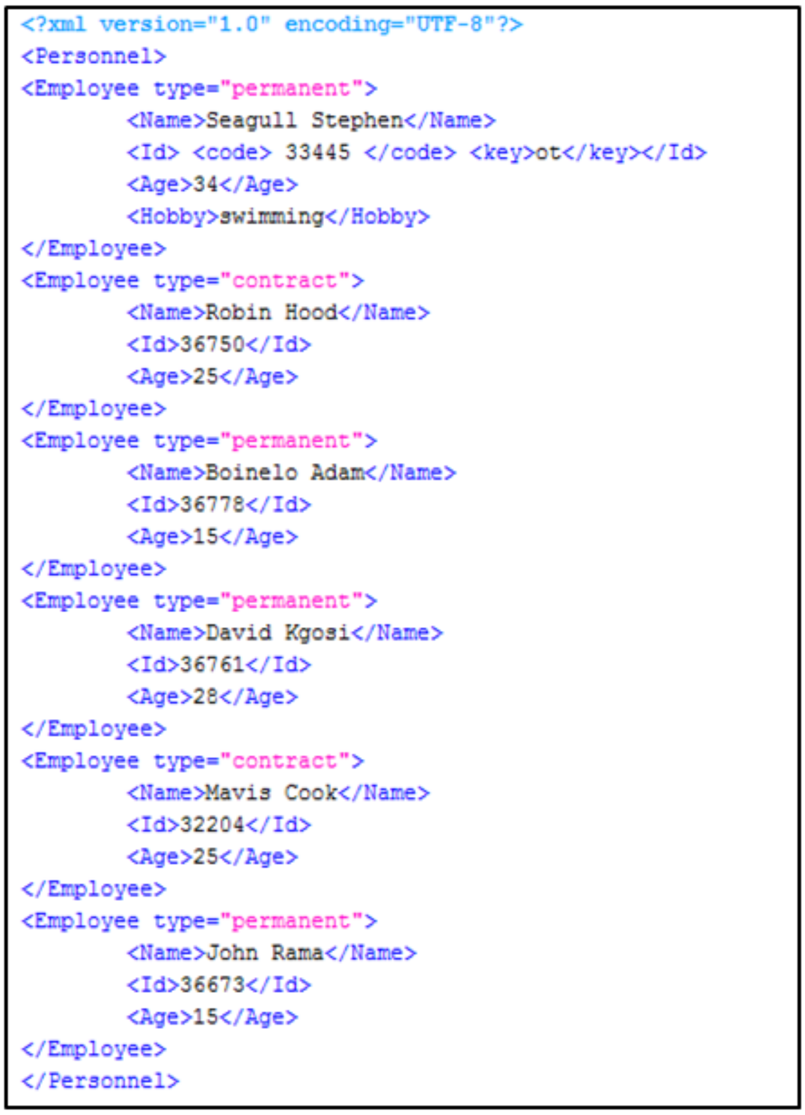
This chapter presents the two components of SIQXC that run on a resource rich environment as shown in Figure 7.2 in the previous chapter; the compressor and decompressor. These are discussed with examples in this chapter. For illustrations, the XML file snippet in Figure 8.1 is used to show how the SIQXC compressor encodes and groups elements to compress the XML file and preserve the structural relationships among nodes. Section 8.2 discusses the compressor and section 8.3 describes SIQXC's decompression process.

8.2 The Compressor

As mentioned in Chapter 5, there is always an inverse relationship between the compression ratio and query support in XML compressors. The challenge therefore is to develop a method that strikes a balance between the two. SIQXC is designed to achieve the highest compression possible while preserving relationships among XML nodes because these relationships are vital for efficient query processing. The way data is stored significantly affects the way it is retrieved [Zhang and Ozsu, 2010, Wong, Lam and Shui, 2007]. SIQXC follows the design in Figure 8.2 shown below to compress XML preserving structural relationships between nodes which will later facilitate efficient query processing in a resource limited environment.

The structural relationships among XML nodes are Parent-Child (P-C), Ancestor-Descendent (A-D) and Sibling relationships [Haw and Lee, 2008, Haw and Lee, 2009]. These relationships are in some contexts referred to as axes or edges [Zhang and Ozsu, 2010].

Identifying structural relationships among nodes is a core process in query processing [Yun and Chung, 2008, Gou and Chirkova, 2005, Lu et al., 2004, Jiang et al., 2004, Chen et al., 2005, Chen et al., 2006, Zhu et al., 2008, Xu et al., 2009, Jiang et al., 2009, Lu et al., 2011] therefore they need to be preserved. The following discussion demonstrates how this compression method preserves these relationships.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<Personnel>
<Employee type="permanent">
        <Name>Seagull Stephen</Name>
        <Id> <code> 33445 </code> <key>ot</key></Id>
        <Age>34</Age>
        <Hobby>swimming</Hobby>
</Employee>
<Employee type="contract">
        <Name>Robin Hood</Name>
        <Id>36750</Id>
        <Age>25</Age>
</Employee>
<Employee type="permanent">
        <Name>Boinelo Adam</Name>
        <Id>36778</Id>
        <Age>15</Age>
</Employee>
<Employee type="permanent">
        <Name>David Kgosi</Name>
        <Id>36761</Id>
        <Age>28</Age>
</Employee>
<Employee type="contract">
        <Name>Mavis Cook</Name>
        <Id>32204</Id>
        <Age>25</Age>
</Employee>
<Employee type="permanent">
        <Name>John Rama</Name>
        <Id>36673</Id>
        <Age>15</Age>
</Employee>
</Personnel>
```

Figure 8.1: XML document snippet

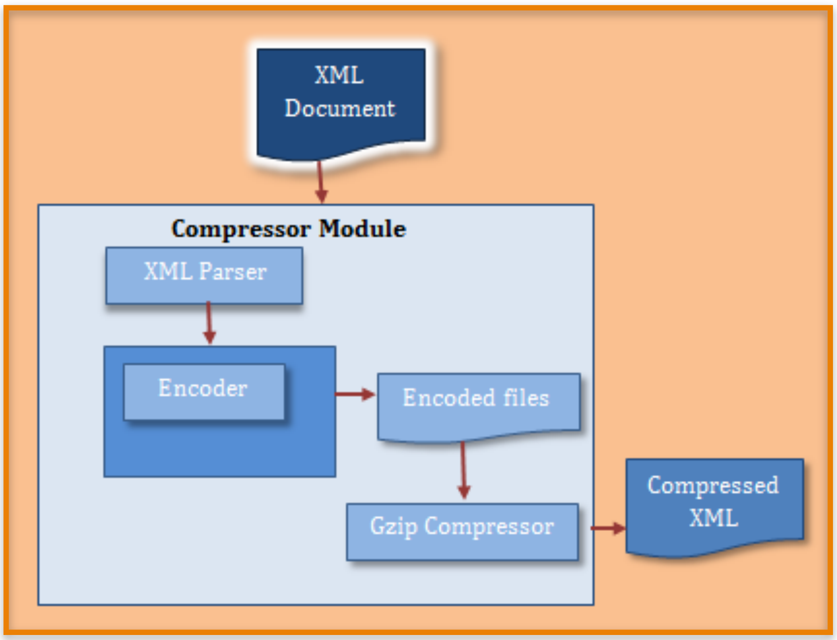


Figure 8.2: The compressor design

8.2.1 Compression Process overview

SIQXC uses four processes to compress an XML document; parsing, labelling, clustering and *gzipping*. Two of these processes, labelling and clustering, are run concurrently. Labelling and clustering keeps track of structural relationships among nodes thereby enabling efficient query processing on compressed data. Prior to labelling and node clustering the XML document is parsed using Document Object Model (DOM) to create a tree representation.

8.2.2 XML Parsing

SIQXC uses DOM to parse the XML file. It was indicated in Chapter 2 that DOM uses a lot of memory but this is not a problem in this case because the compression part of SIQXC is run in a resource rich environment. Also SIQXC deals with trees that are relatively small given that the size has to be one that can be compressed to fit in a resource limited devices so memory use during compression is not a concern. Unlike SAX, DOM preserves relationships [Lam et al., 2006] so its tree representation makes labelling much easier that having to handle SAX events.

SAX is best for streaming application whereas DOM is suitable for databases [Lam et al., 2006]. The XML snippet in Figure 8.1 is parsed using DOM resulting in a tree representation of element nodes shown in Figure 8.3.

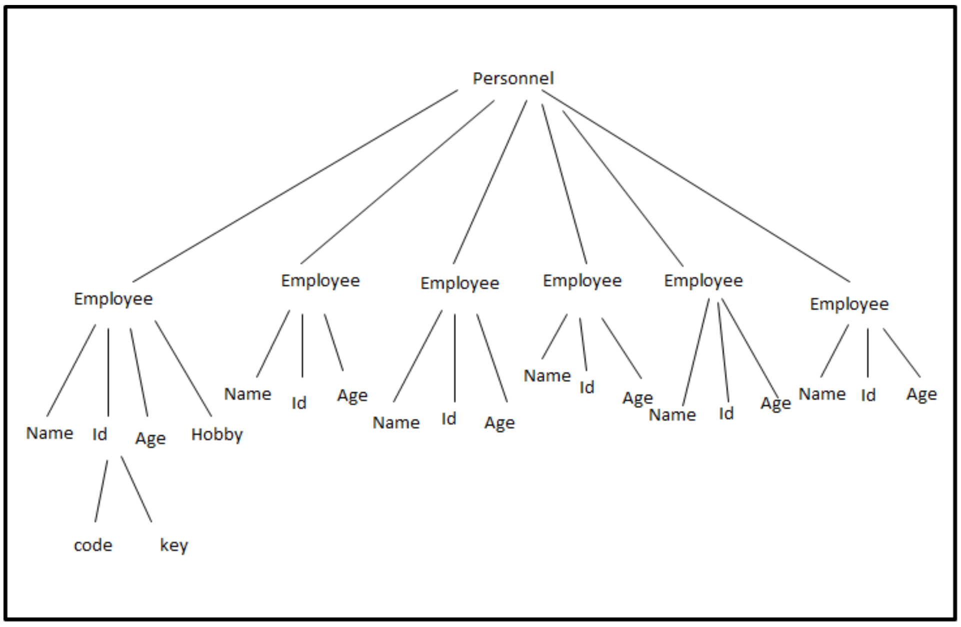


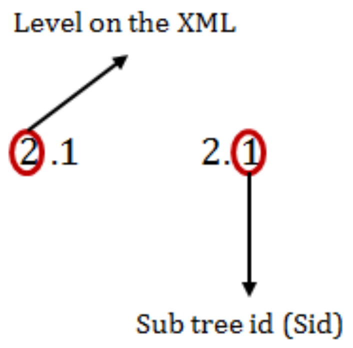
Figure 8.3: XML tree representation

8.2.3 Encoding

After parsing XML the compressor encodes the XML tree obtained. Labelling has been used in XML to create structural summaries [Weigel et al., 2005] that optimize query processing by pruning the XML tree but maintaining structural relationships. It is said that they are better than indexing schemes since in most cases they require less storage space [Li and Ling, 2005, Khaing and Thein, 2006]. This work adopts a novel simple 2-tuple integer encoding system (level, Sid) to label the XML tree. The first part of the label indicates the level of the node on the XML tree and the second part indicates the identity of a sub tree (Sid) it belongs to. In SIQXC only element nodes are labelled. This labelling is achieved by dividing the XML tree into sub trees on the second level. Each sub tree is given a unique code (Sid) by which all its children and descendants are identified.

Essentially each child of the root node (nodes in level 2) is allotted this code. These labels ensure easy determination of structural relationships among nodes during query processing. The following discussion shows examples of how these relationships are retained.

To illustrate this encoding process, consider a node label **2.1**. From this label we derive that the label belong in level **2** of the XML tree and that this node belongs to sub tree **1** (see the example below).



All children of this node will inherit **1** because they belong to the same sub tree as their parent but the level will increase by one to **3** resulting in node label **3.1** thereby preserving the P-C relationship. Since all children for node **2.1** have the same code which is **3.1** the sibling relationship is also maintained. Notice that a node in the same level but a different sub tree will have a code such as **3.2** indicating that though it is in the same level it is not a sibling of the **3.1** nodes because it belongs to a different sub tree. As the labelling continues deep into the sub tree the level increases but the *Sid* stays the same. All children of the **3.1** node are assigned label **4.1**. A node with label **4.1** is a descendent of a node with label **2.1** hence the A-D relationship preserved. Preserving these structural relationships enables SIQXC to support a wide range of queries efficiently.

Figure 8.4 demonstrates a SIQXC encoded tree representation of the XML tree in Figure 8.3.

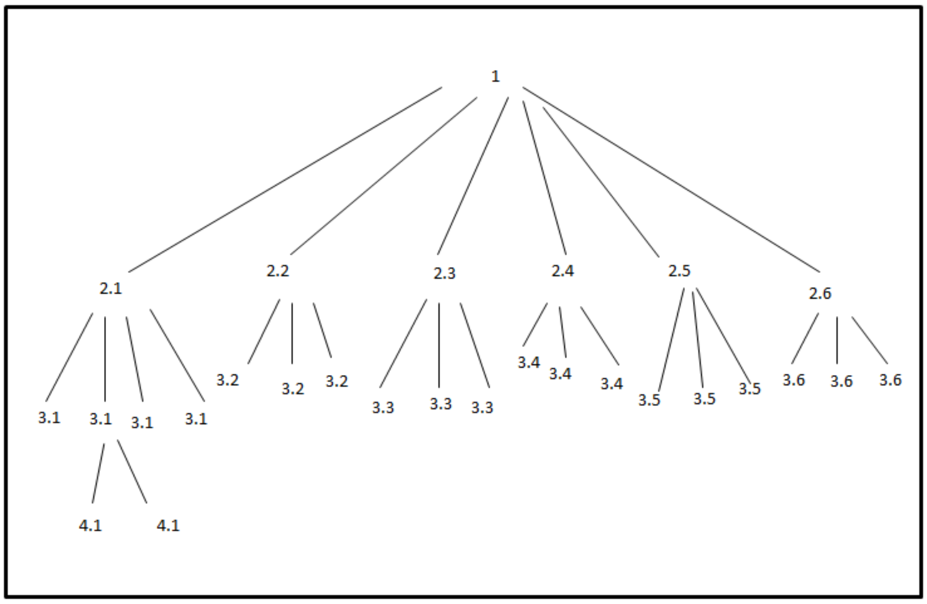


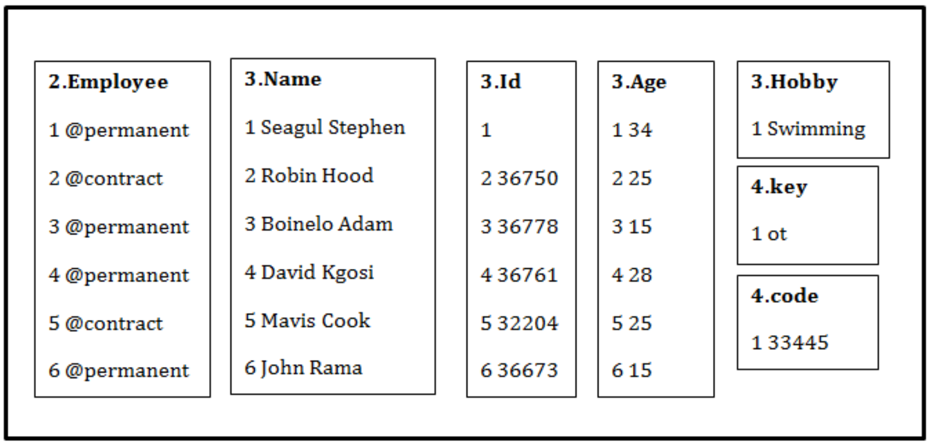
Figure 8.4: Encoded XML tree representation

This level based encoding system is simple yet effective. It does not require a lot of storage space like the prefix labelling schemes [Tatarinov et al., 2002, O'Neil et al., 2004, Cohen et al., 2002] neither does it involve complex computations in creating labels like the prime number labelling schemes [Wu et al., 2004]. However, sibling order is lost during the labelling process. Sibling order is not of great importance in this work because it is rare for users to require this in their queries, instead they use tag names which are preserved. Updates are not supported as discussed in Chapter 9 so it is assumed that the compressed file would not need to be decompressed. The encoding used is therefore sufficient for this work.

8.2.4 Clustering

Since siblings share the same label, their tag names are used to differentiate them during the clustering process. During the labelling process the attributes and text nodes are linked with their respective element nodes by the Sid. This process clusters elements by their tag name and level. Each container is named after the tag name and level of the elements it contains. The container stores the attributes and text nodes along with their Sid. This clustering removes the redundancy in the XML structure thereby significantly reducing the file size. A similar approach was used in NCIM [Liao et al., 2010] but their labelling is different in that creates a 3-tuple label.

The resulting containers are each named after the level and element name as shown in bold in Figure 8.5. In reference to the XML snippet in Figure 8.1**,** all data (text and attributes) associated with the **Name** elements is stored in an encoded container called **3.Name** because this node is in level **3**. Each text node associated with an element is stored alongside the Sid of the sub tree it belongs to. Consider Figure 8.5 for a full representation of all containers created by the SIQXC compression process of the XML document in Figure 8.1.



2.Employee	**3.Name**	**3.Id**	**3.Age**	**3.Hobby**
1 @permanent	1 Seagul Stephen	1	1 34	1 Swimming
2 @contract	2 Robin Hood	2 36750	2 25	
3 @permanent	3 Boinelo Adam	3 36778	3 15	
4 @permanent	4 David Kgosi	4 36761	4 28	
5 @contract	5 Mavis Cook	5 32204	5 25	
6 @permanent	6 John Rama	6 36673	6 15	

4.key
1 ot

4.code
1 33445

Figure 8.5: XML Data grouped into containers

8.2.5 Back end compression

After clustering nodes, text and attributes into containers the containers are passed to a gzip [Deutsch, 1996] compressor. This compressor is based on the DEFLATE algorithm which combines the Lempel Ziv 77 (LZ77) and Huffman coding [Deutsch, 1996, Feldspar, 1997, Salomon, 2007]. LZ77 is a lossless dictionary based algorithm in which strings of characters are mapped to a single code. Huffamn coding on the other hand uses variable length code tables to encode the source data. DEFLATE uses LZ77 to find repetitive characters and builds coding tables using Huffman coding. Gzip concludes the compression process of SIQXC resulting in a compression ratio that would allow a large XML file to some extent be stored and manipulate in a resource restricted device like a smartphone. This would give the user local access to as much of their data as possible through the SIQXC query processor dicusssed in Chapter 9.

8.3 The Decompressor

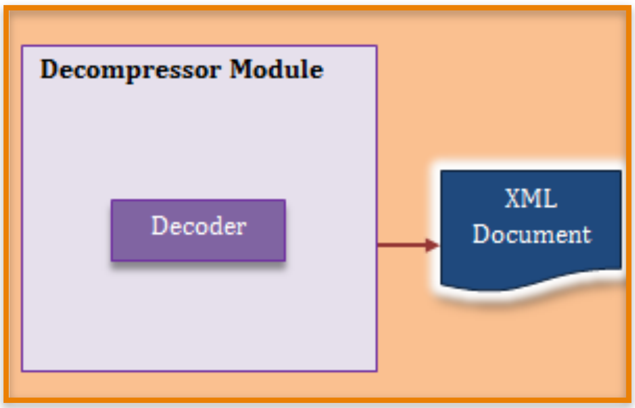


Figure 8.6: The decompressor design

The SIQXC decompressor decompresses data by considering the level that associated with each container. As discussed during compression, nodes are clustered according to their name and level, so each container contains text and attributes from elements with the same name and level but different sub trees.

To receate the XML file, each node label is replaced by the tag name which forms the opening and closing tags of that element. The data associated with each element is then put between the opening and closing tags unless it is an attribute. Attributes are identified with the **@** symbol. In case an attribute is encountered, it is added to the opening tag of the element. This process is done in ascending order of *Sids* in each level to form the XML tree's fanout. From Figure 8.5, the first container to be decompressed is the **2.Employee** container since it has the lowest level number. Each label is then replaced by the node name **Employee** derived from the container's name. Attributes and their values are then assembled with their respective **Employee** nodes. The process is repeated for all containers and we finish with the original XML file. As mentioned sibling order is lost during compression so the resulting XML file will likely have siblings in a different order from the original XML tree. Sibling order is beyond the scope of this research as discussed above.

8.4 Chapter Summary

Chapter 8 presented the compressor and decompressor discussing the processes involved in each component with illustrations. SIQXC uses a novel simple 2-tuple integer encoding system to labelling XML after parsing it with DOM. The labelled nodes are clustered according to their tag name into containers to remove repetitive structures. Individual containers are further compressed using a DEFLATE algorithm based compressor gzip, which combines LZ77 and Huffman coding. The next chapter discusses the query processor.

CHAPTER 9

THE QUERY PROCESSOR

9.1 Introduction

Techniques for extracting data from XML documents is an important issue of XML research [Liu and Ling, 2002]. As discussed in Chapter 2 the way data is stored influences the way it is extracted. This also determines the efficiency of retrieving the data. This chapter introduces a SIQXC query processor that extracts data from SIQXC compressed data. SIQXC supports a wide range of XPATH operators and this is demonstrated in Section 9.3 and its subsections. The chapter gives a general overview of XML query processing leading to a detailed discussion of SIQXC query processing.

9.2 XML Query Processing Overview

Query processing in XML depends heavily on the relationships [Robie, 2007] among nodes therefore users are supposed to be sufficiently aware of the document structure to be able to extract the information they need. The user's unfamiliarity with the document structure may lead to misunderstandings during query processing. Such slight misunderstanding may lead to unsatisfiable queries that render empty answers [Brodianskiy and Cohen, 2007]. This is a requirement for the SIQXC query processor too. Structural relationships are necessary to fetch the relevant data according to the input query. This query processor supports a wide range of XPATH operators. In XML query processing, a query should define the **information** that the user is looking for, the **scope** through which it is to be found and the **context** in which it should be presented [Schlieder, 2002]. An XPATH query does this by defining a way to navigate an XML tree to return a specific set of nodes reachable through the path in the expression [Flesca et al., 2003].

$$\underbrace{\text{Personnel/Employee [Age>20]}}_{\text{Location path}}\text{/}\underbrace{\text{Name}}_{\text{Output expression}}$$

The expression comprises of two parts; a location path (**scope**) and output expression (**information + context**) [Peny et al., 2003]. The location path is a sequence of steps to the desired node whereas the output expression specifies the portion of the matching element that forms the result. A location path is normally made up of axis that represent node relationships and sometimes it include a predicate as demonstrated in the example above.

9.2.1 General XML Query Classification

This section gives an overview of the general XML query classification. The query type determines how a query is evaluated and this also applies in SIQXC query processing as discussed later in this chapter. As discussed in Chapter 2, there are two main types of queries; keyword and structural queries. Structural queries can be further classified as path or twig queries. Path queries involve a simple path from the root that guides the tree navigation to the required data. Twig queries on the other hand can have relative paths and predicates. These are sometimes referred to as branching queries because they do not just follow a simple path. Evaluating such queries sometimes involves the **decompose-match-merge** approach mentioned in Section 3.3.4.1. This approach usually involves intermediate results that are then merged to generate the final result set. This is common where conjunctions and disjunctions are involved. This classification encompasses all other classifications like AlHamadani's criteria, conjunctive and range queries [AlHamadani, 2011] and Schmidt et al.'s classification that is based on the operators or the concept to be tested in the XMark benchmark [Schmidt et al., 2002].

The following sections discuss SIQXC query processing including the query classification that is used. The classification also describes how each query type is evaluated with examples. The illustrations assume the compressed data shown in Figure 8.5 in the previous chapter

9.3 SIQXC Query Processing Overview

The SIQXC query processor is made up of a query analyser and a query search engine. This query processor accepts an XPATH query and passes it to the query analyser where it is analysed to determine the type. The query type dictates the method to be invoked for the evaluation. This approach was used in TwigX-Guide discussed in Section 3.3.3. Each method defines how a query is decomposed and evaluated. Some queries adopt the **decompose-match-merge** approach [Zhang et al., 2001, Chen et al., 2005, Su-Cheng and Chien-Sing, 2009, Bruno, Koudas and Srivastava, 2002] used in twig query processing (see section 3.3.4.1) and others just use the **decompose-match approach.** The method invoked also exerts influence in identifying the compressed containers that are necessary to evaluate the query. The query is then evaluated by the query search engine rendering a result set.

An XPATH query is accepted as a string in SIQXC. This string is decomposed into sub strings according to the invoked method. The query is decomposed into sub strings that when evaluated would give the same result set as the original XPATH query if it were evaluated on the original XML document. Figure 9.1 below shows the structure of the SIQXC query processor.

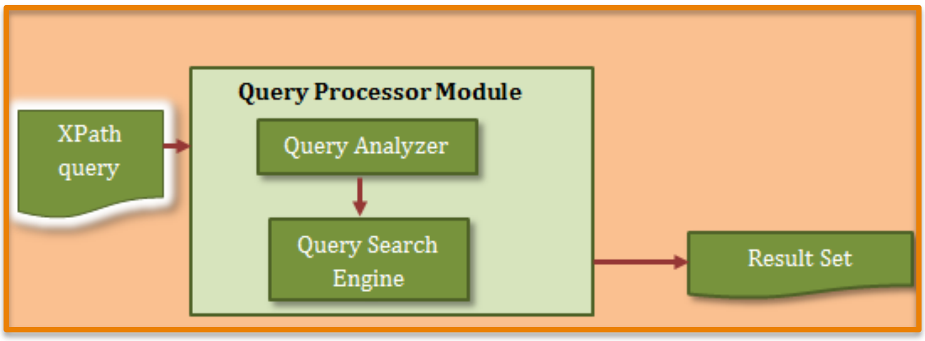


Figure 9.1: The query processor design

As mentioned above the query analyser analyses the query to determine the type to invoke the relevant method for query evaluation and query search engine has different methods that efficiently evaluate each type of query. The next section discusses these query types and how they are evaluated.

9.4 SIQXC Query Classification Overview

The query classification in this work overlaps with other classifications that are mentioned in Section 9.2.1 above but it slightly differs from them because it is specifically tailored according to the query search engine methods in SIQXC. With SIQXC there are two main query types: predicate (**branching**) and non-predicate (**simple path + keyword**).

9.4.1 Non Predicate Queries

Non-predicate queries can either be a **keyword** or a **simple path** query**.** A keyword query involves a single string which is run against the database to bring up all matching strings regardless of their position in the XML tree therefore does not depend on structural relationships among nodes. An example of a keyword query in relation to the XML document shown in Figure 8.1 can be 'Name'. This query returns the values of every 'Name' element in the XML document. Unlike a keyword query, a simple path query is a structural query that depends on relationships among nodes for query evaluation. It specifies the exact path to follow to the required data. This query is composed of steps separated by a single forward slash '/' from the root node to the desired node. An example of this query would be 'Employee/Name' which returns the values of all 'Name' elements that have Employee as a parent.

9.4.2 Predicate Queries

Predicate queries are any queries with a criteria or a regular expression enclosed in square brackets '[ ]' that has to be applied on the specified node or nodes to retrieve a specific subset of data. The criteria can be a disjunction, conjunction range or comparison. SIQXC classifies predicate in two ways. One classification is based on the **output expression** whereas the other is based on the operators in the predicate.

Sometimes a predicate is applied on a node **A** to return results from node **B**. Node **B** in this case being the output expression. This function leads to SIQXC's first classification of predicate queries; **same node predicate** and **divergent node predicate** query. Every criteria, conjunctive and range can have same node or divergent node predicates which is why this work follows a different query classification from AlHamadani's [AlHamadani, 2011]. In this work's classification the same node predicate applies the predicate on node **A** and returns values of node **A** that satisfy the predicate whereas the divergent node predicate applies the predicate on node **A** but returns values of node **B** that correspond to values of node **A** that satisfy the predicate.

As aforementioned the other classification is based on the number of operators in the predicate; **single operator** and **multiple operator queries**. A single operator query like the name suggests has one operator whereas a multiple operator query has more than one operator. A multiple operator query usually has a two part predicate connected by a conjunction (AND) or disjunction (OR).

9.5 Detailed Discussion Query Types and Their Evaluation

Once the query type is identified by the analyser the relevant method is invoked to evaluate the query. The query type determines how the query is split for efficient processing. The steps followed to evaluate each query type are discussed below with an example to demonstrate how the query processor processes each type of query.

9.5.1 Keyword query

As stated above the keyword query is a query that does not depend on node relationships. Given that containers are named after the element name, evaluating this query involves checking for string containment. If the container's name contains the input string its contents are returned as a result set regardless of its level or position on the XML tree. The string after the level should be equal to the input string ignoring the case. The result set will include all the text data and or attributes. Consider the following example

Q1

```
Age
```

Executing **Q1** returns a list of values in the container 3.Age: (34, 25, 15, 28, 25, 15).

9.5.2 Simple path query

Q2

```
Personnel/Employee/Age
```

With Simple path queries the query processor retrieves part of the XML document according to a general specification which in this case is the path in the input query. However, the query analyser first analyses the path to identify the most important node that is output expression. This is a node that carries the data that the user actually wants. The name of the node usually comes after the last slash '/' in the input query. Identifying this node enables the query processor to only open the relevant containers to retrieve the required data. To process **Q2** which is a simple path query, the query analyser identifies Age as the important node and the query processor returns the same result set as that of **Q1.**

9.5.3 Same node predicate query

The same node predicate query retrieves data according to a specific criterion which can sometimes be a regular expression. In SIQXC processing same node predicate queries begins with identifying which container to retrieve as with the simple path query. The query is decomposed to get the criteria or filter and the output expression. The string enclosed in the square brackets '[]' that form the predicate is extracted to identify the operator and its position (**x**). The name of the container on which the predicate is to be applied is found by taking a substring of the given predicate from length **0** to length **x-1.**

Using the example in **Q3**, 'Age>20' is the predicate. The position of the operator, **x**, is **4** (x=4). Taking the substring of the predicate at position **(0, x-1)** which is **(0, 3)** gives 'Age' as the name of the container to apply the predicate to. Having identified the container the predicate is applied to return the relevant data. The predicate is therefore applied to container '3.Age'. Unlike in **Q1** and **Q2** the query processor only returns the values that meet criterion. Notice that for **Q1** and **Q2** the query processor returns all values in the container. For **Q3** only (34, 25, 28, 25) are returned because they are greater than **20** which is the filter specified in the predicate.

Q3

```
Personnel/Employee [Age>20]/Age
```

9.5.4 Divergent predicate query

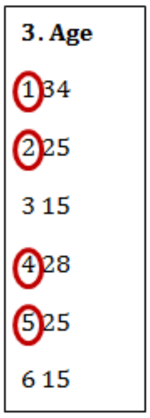
Processing the divergent predicate query uses the same technique that is followed in the same node predicate query evaluation except in this case the query processor does not return the list retrieved from applying the predicate on the specified node as the final result but rather treats that list as the intermediate results.

Q4

```
Personnel/Employee [Age>20]/Name
```

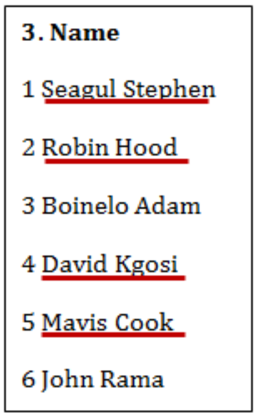
The query processor uses the Sids associated with each entry in the intermediate results to retrieve relevant data from the node in the output expression. As with processing *simple path* queries this node is the node that comes after the last '/'. The SIQXC query processors returns the data from anyone of the nodes that has an *Sid* that matches one in the intermediate results. The retrieved data is returned to the user as a final result. To process **Q4** shown above, the query evaluation follows the same process that was used to evaluate **Q3**.

Following this process container '3.Age' is retrieved. Applying the predicate on it yields a result set (1, 2, 4, 5) as intermediate results. Note that these are Sids or identifiers not values. These are the Sids that corresponds with 'Age' values (34, 25, 28, 25) respectively. In the next step all values in container '3.Name' with the above Sids are extracted. The values that make up the final result set therefore are: (Seagul Stephen, Robin Hood, David Kgosi, Mavis Cook) as shown in Figure 9.2A and 9.2B:



3. Age
1 34
2 25
3 15
4 28
5 25
6 15

Figure 9.2A: Container 3.Age



3. Name
1 Seagul Stephen
2 Robin Hood
3 Boinelo Adam
4 David Kgosi
5 Mavis Cook
6 John Rama

Figure 9.2B: Container 3.Name

9.5.5 Single operator query

A single operator query is a predicate query that only has one operator. **Q3** and **Q4** above are examples of a single operator query. To evaluate this query the predicate is applied on the specified container to get the final or intermediate results as described above

9.5.6 Multiple operator query

A multiple operator query is a query that has more than one parts separated by an operator or operators. This type of queries is the one that use the *decompose-match-merge* approach used in twig query processing described in Chapter 3. As with other predicate queries the string enclosed within the '[ ]' is extracted. In this case the string is decomposed according to the operators yielding two or more parts that are then evaluated separately. The intermediate results are then merged to give a final result depending on the operators used. These queries normally contain a disjunction or conjunction of two or more specific criteria. The conjunction returns an intersection (∩) of the results whereas the disjunction returns a union.

With a conjunctive query the query processor evaluates the part of the predicate that comes before the conjunction to get intermediate results. The second part of the predicate is only evaluated on nodes that have a matching Sid as the one in intermediate results instead of the whole container. If the first evaluation returns an empty set, the second part is not evaluated instead an empty set is returned as the final result because as mentioned above a conjunction returns an intersection (∩).

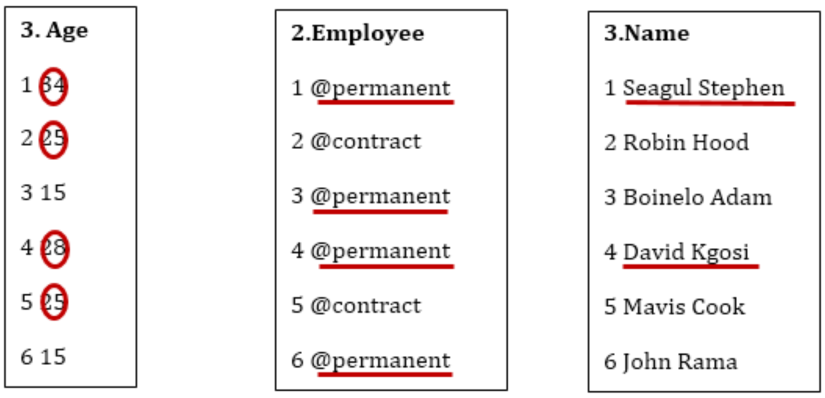
As for the disjunction all parts of the predicate are evaluated returning the union (∪) of all intermediate results. These results are merged to give a final result.

To process the conjunctive query, **Q5**, shown below the query processor removes the square brackets around the query and splits the predicate into two predicate **A** and **B**. This is done to identify the containers that are necessary to evaluate the query. As with other queries above the predicate is treated as a string. The position of the AND operator is identified. Predicate **A** is therefore from 0 to the beginning of the AND operator and predicate **B** is from the end of the AND operator to the end of the string. Having split the predicate the necessary containers are identified following the process described in processing a same node predicate query in section 9.5.3.

Q5

```
Personnel/Employee [Age>20 AND @type='permanent']/Name
```

Following that process for **Q5**, the necessary containers are '3.Age' for predicate **A** and '2.Employee' for predicate **B**. The query seeks for a child node 'Name' of an 'Employee' with attribute 'permanent' and 'Age' greater than 20 so container 3.Name is also necessary. Applying the predicate on container '3.Age' yields (34, 25, 28, 25). If their corresponding Sids (1, 2, 4, 5) exist in '2.Employee' the **B** predicate is applied on the attribute resulting with Sids (1, 4). Names corresponding to these Sids in the '3.Name' container are then retrieved resulting with (Seagul Stephen, Mavis Cook).



3. Age	
1	34
2	25
3	15
4	28
5	25
6	15

2.Employee	
1	@permanent
2	@contract
3	@permanent
4	@permanent
5	@contract
6	@permanent

3.Name	
1	Seagul Stephen
2	Robin Hood
3	Boinelo Adam
4	David Kgosi
5	Mavis Cook
6	John Rama

Figure 9.3: Container 3.Age, 2.Employee and 3.Name with selected results

Q6

```
Personnel/Employee [Age>20 OR @type='permanent']/Name
```

Q6 is very similar to **Q5** except that it uses a union instead of an intersection due to the presence of the OR operator rather than the AND operator. Processing **Q6** therefore follows the same process as **Q5** except predicate **B** is applied on the 2.Employee even if corresponding Sids from the results of applying predicate **A** do not exist in container 2.Employee because the final result is a union. The final result for this query yields (Seagul Stephen, Robin Hood, David Kgosi, Mavis Cook, John Rama) which is the union of the sets of results retrieved from applying the predicates on the respective containers.

9.5 Summary

SIQXC query processor evaluates queries by identifying their type and invoking the type specific method. The type of queries determines how the query is split from which the name of the relevant container is derived. In the case of a predicate query, a predicate is applied on the named container otherwise all its contents are returned as a result. Sometimes evaluating a query may results in intermediate results. These intermediate results are then used to retrieve the final results by getting values that have Sids that match the ones in the intermediate results. This query processor handles wide range of XPATH operators efficiently as shown from the above discussion making it a novel approach to handle compressed XML in a resource limited environment.

9.6 Chapter Summary

The way data is stored influences the way it extracted. This chapter covered the way SIQXC extracts data from a SIQXC compressed XML. It has outlined each step that is taken to evaluate the supported XPATH operators. The SIQXC query classification has been explained to demonstrate how this query processor processes each type of queries. Though SIQXC supports a wide range of queries it does not support position, closure and aggregation given that the query processor is run on a resource limited device. These types of queries pose significant challenges to query processing especially closure [Penny et al., 2003]. They therefore require complex lines of code to process hence they are not supported in SIQXC because they would require a lot of space to evaluate. Supporting these types of queries would be an extravagance since they are expensive and rarely used. This work is intended to be a proof of concept and supports a reasonably wide range of queries that a user would otherwise not be able to execute in their smartphones in the absence of the Internet.

CHAPTER 10

EXPERIMENTAL DESIGN

10.1 Introduction

Chapters 7, 8 and 9 presented SIQXC and its components providing basic notions about procedures that are performed within each component to support the hypothesis and achieve goals outlined in Chapter 6. This Chapter presents several experiments that were designed to test the different units of SIQXC. Each component was tested for its functionality and performance. This experimental framework is used to empirically test SIQXC. The experimental design explains the objective of each experiment outlining the properties that are tested for in each component. Table 10.1 shows a summary of the components and the properties that this experimental framework tests.

The environment in which tests were run in is also specified. Recall that, the components of SIQXC run in two different environments. The compressor and decompressor run in a resource rich environment that is specified below whereas the query processor runs in a resource constrained environment. Testing for functionality ensure that each component achieves its goal. For the compressor the compression ratio and compression time are measured whereas for the query processor the query response time is measured and the type of queries it supports are established. The decompressor was not evaluated in this work since as discussed it was only included in the design for completeness but it is not necessary because SIQXC does not support updates. Several XML benchmarks and datasets are explained in this chapter as well.

10.2 A Description of the Evaluated Factors

This section describes the factors that are tested to evaluate each SIQXC component.

SIQXC Component	Tested Factors
Compressor	- Compression Time - Compression Ratio
Query Processor	- Query Response Time - Functionality

Table 10.1: Tested factors on the SIQXC components

Compression Time: Compression Time (CT) is the time taken to compress an XML file. This is measured in seconds. The relationship between compression time and the file size is also observed.

Compression Ratio: Compression Ratio (CR) measures the difference between the original XML document and the compressed file using the equation below:

$$CR = 1 - \left(\frac{\text{size of compressed}}{\text{size of original file}} \right)$$

Functionality: The functionality test for the query processors determines the types of queries that are supported.

Query Response Time: Query Response Time (QRT) measures the time taken in seconds for the query processor to evaluate different types of queries. As with compression time the relationship between query response time and the file size is also measure.

10.3 The Experiments

The next three sections discuss the experiments that were carried out and the factors that were being investigated in each of them.

10.3.1 The Compressor Experiment

The main objective of this experiment is to establish the average compression ratio (CR) and the average compression time (CT) of SIQXC. As previously stated in Chapter 8 an XML file is labelled with a 2-tuple integer encoding system used by this compressor and nodes are grouped into containers. Recall that in Chapter 6 it was stated that the aim of this compressor is to investigate the extent to which this process can compress a file such that it can be stored and processed in a resource limited device. File sizes are recorded before and after compression to calculate the compression ratio.

The pruning performed in the compression process is expected to significantly reduce the file size resulting in a competitive compression ratio. This is followed by applying gzip to the individual containers and recording their overall size. Different sets of XML data are used to observe the behaviour that the compressor exhibits with different files. The datasets used in the experiment are different in terms of the tag to data ratio, the depth and breadth of trees.

The compressor can be expected to behave differently with each data set. Given its pruning nature it should show a better compression ratio with trees that have a high tag ratio over data. This is because more tags mean a lot of redundancy which is removed by encoding and clustering elements. A tree with repeated sub trees across the breadth will also result in high compression ratio due to grouping. Compression ratio before applying gzip is also recorded to measure the effectiveness of the the encoding and clustering process.

10.3.2 The Query Processor Experiment

The query processor experiment measures performance and functionality covering different types of queries and how their complexities affected response time and accuracy. Functionality tests the ability of the query processor to effectively process each type of query giving accurate results. Several queries are run on the query processor to determine the types of queries it supports. Having established the type of queries the SIXQC query processor supports, the queries are run again to find out the query response time.

The query response times are expected to be different. Complex queries like the multiple operator queries (disjunction and conjunction) are anticipated to take longer than other queries. Same node predicate queries are expected to be processed faster than divergent node predicate queries. Keyword queries on the other hand can only have a better response time in a case where there is only one container that has the keyword. If there are many containers with the keyword then processing is prolonged because all data in these containers should be displayed. The behaviour of the query processor in relation to each query was observed by running the different queries on the same compressed XML file.

The query processor was developed in two phases. In the initial phase it was developed to run on a resource rich environment and then modified to run on a resource constrained environment. At the time of writing all query type were tested on a resource rich environment but only two types were run on the resource restrained environment.

10.4 Implementation of the SIQXC Prototype Overview and Test Environments

10.4.1 The Prototype Implementation

The compressor and the query processor were both implemented using the JAVA programming language (JDK1.7.0).

10.4.1.1 The compressor

As discussed in chapter 8 section 8.2.2 SIQXC parses an XML using DOM. In this implementation of the compressor, the Document Object Model (DOM) which is a component API of the JAVA API for XML processing is used to parse the XML document resulting in a tree that is then labelled using a recursive function followNode() which accesses the tree in depth first pre order traversal. During the labelling process the data and attributes values are fetched using getPCDATA() and getNodeValue() methods respectively and stored in a text file alongside the corresponding Sid for the node in question. The text files created in this process are passed on to a class that is based on the JAVA ZIP package that provides classes for reading and writing the standard GZIP file formats

10.4.1.2 The Query Processor

The SIQXC query processor is implemented in two stages; the generic and platform specific implementations. In the first stage a generic JAVA implementation that can be adapted to specific platforms is implemented, followed by a platform specific implementation, in this case Android. The two implementations are discussed in section 10.4.1.2.1 and section 10.4.1.2.2 respectively.

10.4.1.2.1 Generic Implementation

In this implementation the query processor prompts a user to enter a query via the command prompt. The query analyser checks for square brackets to determine whether the query is a predicate or non-predicate query. If a query is predicate further tests are done to check if it is multiple or single operator query. Having identified the query the right method is invoked. With each method in this implementation the query processor identifies and decompressed the containers that necessary to evaluate a given query if there be any. Decompressing a container creates an unzipped file but it leaves the zipped container intact. The decompressed files are deleted after evaluating each query.

10.4.1.2.2 Android Implementation

For testing purposes the generic implementation was adapted to Android. Android was chosen because it is one of the two most widely used mobile platforms together with iOS [Goadrich and Rogers, 2011] and it supports JAVA 6. This implementation was done in Eclipse IDE with the Android SDK plugin. Although it supports a big subset of JAVA, Android does not run on the JAVA virtual machine but instead has its own called Dalvik [Yang, Chu and Tsaur, 2010; Goadrich and Rogers, 2011; Sharma, 2011; Kim, Agrawal and Ungureanu, 2012]. With Android projects are compiled and run in the Dalvik virtual machine (VM) with each application in the device running inside its VM [Goadrich and Rogers, 2011]. For these reasons some of this implementation is slightly different from the generic implementation. Decompressing necessary containers in this implementation is implied because Android does it automatically on the fly at runtime. All other JAVA 7 methods that are not supported by JAVA 6 like switching on a string are changed. The Android SDK provides a WYSIWYG editor that made it easy to create a simple GUI shown in Appendix II. The user enters a query then clicks execute and the results are printed back on the screen as seen in Appendix II.

10.4.2 Test Environments

As mentioned in Chapter 7, SIQXC runs in two different environments. The compressor and decompressor run on a resource rich environment while the query processor runs on a resource constrained environment. This section outlines the platforms that these sets of experiments were performed on.

10.4.2.1 Resource Rich Platform

This platform is a PC with 3.00 GHz Intel® Core™ 2 Duo CPU E8400 and a RAM memory of 4.00GB. The capacity of the Hard disk of the testing environment is a 148GB. This PC runs a 64 -bit Windows 7 operating system. Profiling was done using the NetBeans IDE 7.2.1.

10.4.2.2 Resource Constrained Platform

The second platform is an Android emulator on Eclipse ADT. A virtual device with specifications of a smartphone was created to run the query processor as an application. The virtual device runs Android 4.4.2 with RAM 512 MiB and an internal storage of 200 MiB. Its CPU is the ARM armebia –v7a. This processor is automatically selected when selecting Android 4.4.2 as the target platform to be supported by the virtual device created.

10. 5 Document Corpus and XML Benchmarks

There is a large corpus of XML documents that contain real data available from multiple sources to use for XML testing and benchmarking. The properties of these datasets and benchmarks are presented in this section.

10.5.1 XML Datasets

Real world datasets that are available differ in structure and size. Some documents are more structural (data centric) while others are more textual (document centric). These properties affect the performance of XML tools. Each dataset is discussed below with its properties. A few of these datasets were used in this experimental framework to evaluate SIQXC.

- **XMark:** The XMark dataset is essentially XML documents modelling an auction website [Schmidt, 2001]. These documents are created through a tool called xmlgen. This XML data generator was developed inside the XMark Project. It accepts a parameter (-*f*) to produce different sizes of XML documents.

- **DBLP**: This is a database of bibliographic information obtained from major Computer Science journals and conference proceedings. The acronym stands for **D**igital **B**ibliography **L**ibrary **P**roject [Ley et al., 2005].

- **TreeBank:** An XML file of parsed English sentences from a Wall Street Journal [Marcus, Marcinkiewicz and Santorini, 1993]. It is considered a complex database for its deep recursive structure. The database is partially encrypted to protect copyright for text nodes.

- **SwissProt:** SwissProt is a curated collection of protein sequence that describes the DNA sequences [Boeckmann et al., 2003]. The description includes the function of a protein, its structure, post-translational modifications and variants.

- **NASA:** This is an astronomical database that contains genuine astronomical data made available to the public as an XML file. The data is converted from a legacy flat file. NASA is used evaluate many XML application that process XML queries [Cover, 2000].

- **Shakespeare:** A collection of marked-up Shakespeare's play as an XML file [Bosak, 1999].

- **Mondial:** Mondial is an XML file containing basic statistical information on countries of the world. This geographic information is an integration of collections from several sources including the CIA World Factbook, the TERRA database and the international Atlas [Suciu and Miklau, n.d.].

10.5.2 XML Benchmarks Review

This is a review of the most popular and widely used XML benchmarks. Benchmarks provide means to evaluate the performance of XML databases by specifying meaningful and relevant task that should be carried out to assess these tools. They offer support for comparative analysis.

XMark: As described on the datasets, XMark models an auction website. This benchmark is a result of the XMark benchmarking project led by a team at CWI [Schmidt et al., 2002; Barbosa, Manolescu and Yu, 2009]. XMark is widely used to evaluate XML applications [Arion et al., 2004; Lu et al., 2005; Barbosa, Manolescu and Yu, 2009]. Its workload includes twenty queries [Barbosa, Manolescu and Yu, 2009] that cover the essentials of XML query processing [Li, n.d]. These queries were evaluated on Monet XML database, an internal research prototype, to give a first baseline [Li, n.d].

XMark queries include simple node selection, navigational queries, document queries where order information is relevant and the computation of other operations like aggregation, sorting, reconstruction and joins [Barbosa, Manolescu and Yu, 2009; Li, n.d]. This benchmark includes a data generator that is free to download from the XMark project website [Schmidt, 2001]. As mentioned above the data generator accepts a parameter that sets the file size of the XML file generated. This gives different users the flexibility of creating XML files of the size they desire. This property is very useful for testing the scalability of an XML tool.

XPathMark: This is an established benchmark that is usually used to evaluate tools that support XPATH queries [Pena, 2013]. It is an XPATH 1.0 benchmark for the XMark document base [Franceschet, 2005]. It was developed at the University of Udine in Italy. This benchmark covers major aspects of the XPATH query language including node tests, references, functions, Boolean operators and different axes. Like XMark, this benchmark also comes with a XML generator that produces XML documents according to a scaling factor specified by the user. It is freely available on the XPathMark website [Franceschet, 2005]

The Michigan Benchmark: The Michigan benchmark was developed at University of Michigan [Runapongsa et al., 2006]. It is an XML Micro-benchmark that consists of a single synthetic document that does not resemble a typical document from any real world application domain. It is mainly designed to cover extensive XML query operations such as: selection of nodes based on predicates over their parents, computing aggregates, updating, joins, matching attributes by value and evaluation over positional predicates [Barbosa, Manolescu and Yu, 2009] through thirty one queries. The file has a depth of sixteen levels and a changeable breadth. This benchmark was applied by the authors to three database systems two of which were native XML databases and one commercial ORDBMS [Barbosa, Manolescu and Yu, 2009].

XMach-1: The XMach-1 benchmark was developed at the University of Leipzig in Germany [Böhme and Rahm, 2001]. This is a multi-user benchmark designed to test a database management system which includes a query processor and other components unlike most benchmarks that are designed to only evaluate the query processor. In terms of performance, it measures throughput instead of query response time. XMach-1 is made of four main components; application server, XML database, loaders and browser clients. Its workload includes eight queries and three update operations. The file size falls between 2KB and 100 KB. These files are generated from most frequent English words [Barbosa, Manolescu and Yu, 2009].

XBench: XBench was developed at the University of Waterloo [Yao, Özsu and Khandelwal, 2004]. The benchmark can be categorised on two factors; data centric versus text centric or multi document versus single document. Those factors can be combined through the toXgen tool to generate four different types of databases; data centric/single document, text centric/ single document, data centric/ multiple document and text centric/ multiple document. Document size ranges from a few kilobytes to several gigabytes. The workload of this benchmark includes text based search operations and bulk loading [Barbosa, Manolescu and Yu, 2009].

TPoX: This is a commercial benchmark developed jointly by IBM and Intel [Nicola, Kogan and Schiefer, 2007]. Like XMach-1, TPoX aims at evaluating the whole system not just the query processor. It evaluates aspects such as concurrent access to data, updates, XQuery and SQL/XML. This benchmark is based on the industry-standard schema FIXML [Fixtradingcommunity.org, n.d]. The schema is used to control the size of an XML file by defining its depth and breadth.

XOO7: XOO7 [Li et al., 2001; Bressan et al., 2003] is a benchmark based on OO7 [Carey, DeWitt and Naughton, 1993]. The OO7 benchmark dataset and queries are converted to be used in an XML based benchmark. The workload includes twenty three queries that cover search operations. XOO7 maintains the same depth for different sizes of documents. Unlike XMark, it provides pre-set sized documents making it almost impossible to use in testing scalability.

10.6 Comparison with Other Compressors

Although much research has been devoted to XML compression, almost all tools in the literature do not currently have publicly available source code [Sakr, 2009; Pena, 2013]. This creates a problem for comparative analysis. The only results that can be used for analysis are the published results. In this work the published results on compression ratios are used for comparative analysis since they are not dependent on the environment they were run on. Performance on the other hand, largely depends on the environment so any comparisons made using the published results would not be precise unless the environment that they were run on is replicated. However, the performance results of the SIQXC compressor are discussed.

10.7 Comparison with Other Query Processors

At the time of writing the author was not aware of any published work on a query processor that has been designed to run on a resource constrained environment therefore there is no tool that can be compared to the SIQXC query processor both on performance and functionality but as with the compressor's performance result, the query processor's results are also discussed in this work.

10.8 Chapter Summary

This chapter describes the experimental framework that is used to test different components of SIQXC to establish their performance and functionality. For the compressor the compression ratio and compression time are measured and the compression ratio is compared with the compression ratio of other existing queryable and non queryable XML compressors. The functionality and performance for the query processor evaluates whether it can generate accurate results.

SIQXC runs on two different environments that are specified above. The environment a system is run on influences its performance therefore the performance of SIQXC components cannot be compare with published results from the existing compressors and their corresponding query processors because their code is not publicly available.

The XML benchmarks and datasets are also discussed in this chapter to present the options that are available for evaluating XML applications including SIQXC. The next chapter presents results obtained from running experiments designed in this chapter.

CHAPTER 11

RESULTS AND EVALUATION

11.1 Introduction

The previous chapter described an experimental framework used to evaluate SIQXC. Each component was tested for functionality and performance. For the compressor, the compression ratio and compression time were measured while the query processor was tested for functionality and query response time. The functionality tests established the type of query processor it supported while the response time measured the time taken to return results for each query. The query processor is said to support a query only and only if it returns accurate results. This chapter presents and discusses the results obtained from the experiments. It also compares SIQXC with other queryable compressors using published results. Published results were used because the existing compressors do not have their source codes publicly available as stated in Chapter 10. For those that do, their codes could not run on the environment used for testing SIQXC due to compatibility issues. Using published results of compression ratio for comparative analysis gives accurate results nonetheless because compression ratios does not depend on the environment that tests were run on but rather the algorithm used for compression.

11.2 The Compressor

In evaluating the compressor twelve files were used. Of this twelve, eight were generated from the XMark benchmark project using different scaling factors. The other four were the NASA and Mondial datasets described in the previous chapter, the uwm dataset which is the list of University of Washington courses and the customer XML file which is part of TPC-H benchmark; the Transaction Processing Performance Council [Tpc.org, 2001].

TPC-H, was not discussed in the previous chapter because it is not an XML benchmark however, the dataset was converted XML by Zack Ives [Suciu and Miklau, n.d.; Senthilkumar and Arputharaj, 2011] All the four datasets were obtained from University of Washington XML Data Repository [Suciu and Miklau, n.d.]. Table 11.1 below shows all the datasets and their sizes in MB.

Dataset	Size (MB)
XMark 1	0.054
XMark 2	0.091
XMark 3	0.879
XMark 4	1.001
XMark 5	2.275
XMark 6	4.616
XMark 7	9.151
XMark 8	17.982
mondial	1.702
nasa	23.889
uwm	2.228
customer	0.491

Table 11.1: Datasets and their sizes

11.2.1 Compression Ratio

The compression ratio of this compressor was measured for all these datasets giving results shown in Figure 11.1 below. The detailed results are shown in Appendix I.

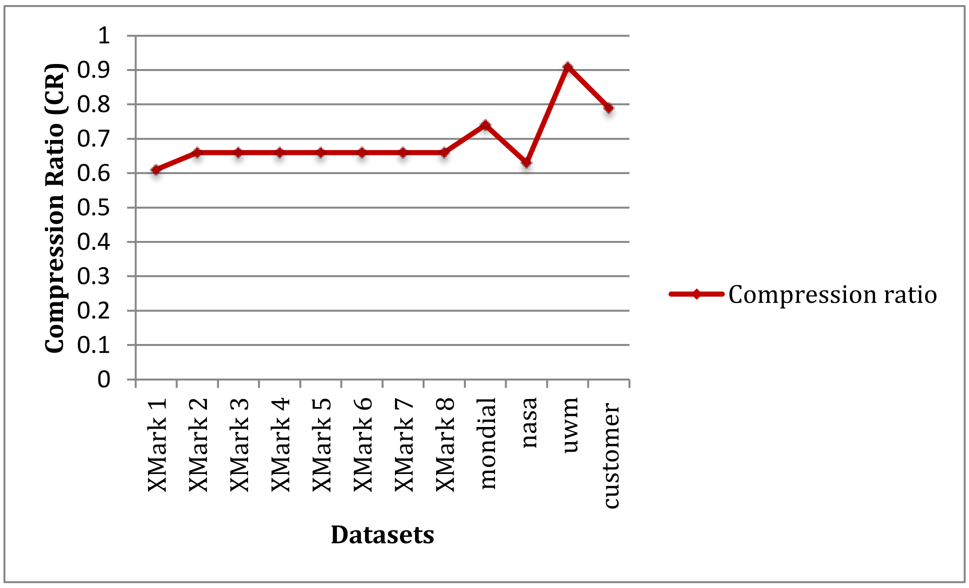


Figure 11.1: Compression Ratio s of different datasets

In the results shown in Figure 11.1 the SIQXC compressor has the lowest compression ratio of **0.61** for the XMark 1 XML file and the highest compression ratio of **0.91** of the uwm dataset. The average compression ratio from these datasets is **0.69**. It is significantly lowered by the XMark file because it is very small. As with other compression algorithms SIQXC does not perform very well with small files. Excluding this result gives SIQXC an average compression ratio of **0.70**. The result can be excluded because compression is normally not necessary when dealing with file as small as that. This file was used to observe how SIQXC performs with very small files. For XMark files the compression ratio remained constant from XMark 2 through to XMark 8. This shows that SIQXC's compression ratio is not affected by the file size but rather the structure of an XML file. The uwm file is larger than the XMark 1 file through to XMark 4 yet the compression ratio achieved with this file is far greater than those achieved with these XMark files. The file size and compression ratio relationship is shown in Figure 11.3.

As mentioned in Chapter 8, the simple 2-tuple integer encoding and clustering of elements achieves some compression. Figure 11.2 below shows the compression achieved by this process against that achieved after further applying gzip. The blue bars show compression without gzip whereas the red bars indicates compression with gzip which is the same results depicted on Figure 11.1. As expected better compression ratios are achieved after using gzip. The simple 2-tuple interger encoding and clustering alone achieves reasonable compression ratio with the highest at 0.58 which is higher than the average compression ratio of the widely known queryable compressors; XPRESS [Min, Park and Chung, 2003] and XGRIND [Tolani and Haritsa, 2002].

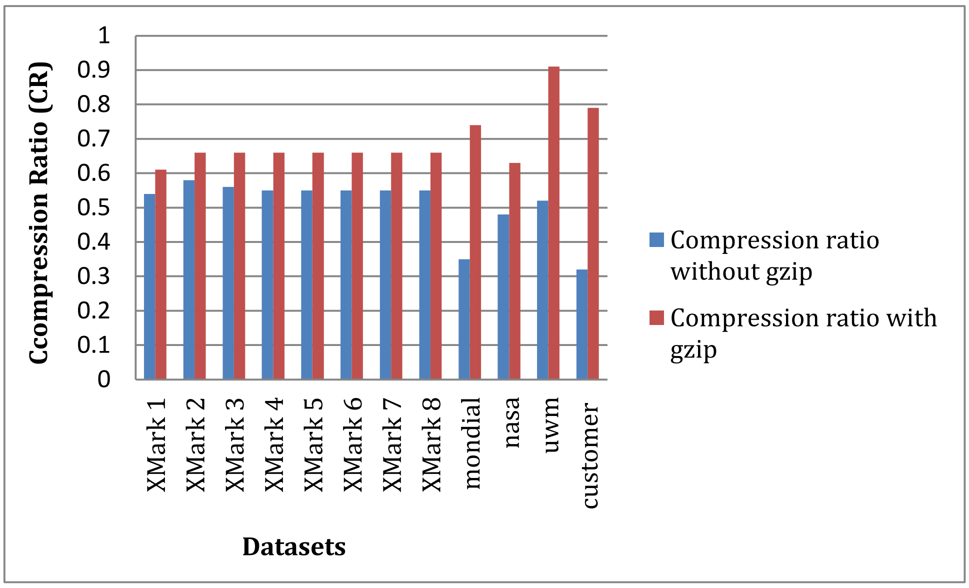


Figure 11.2: Compression Ratio with and without gzip

Compression with gzip makes a significant difference with some files but not all. As shown in Figure 11.2 above it does not make that much difference especially with smaller files. It only makes 7% difference with XMark 1 file but 47% with the customer file. The performance of the gzip does not just depend on size in this compressor but also on the number of containers created during the clustering process. If data is clustered into fewer containers the overall compression ratio is significantly improved. This means the structure of the XML also affects compression ratio achieved by applying gzip.

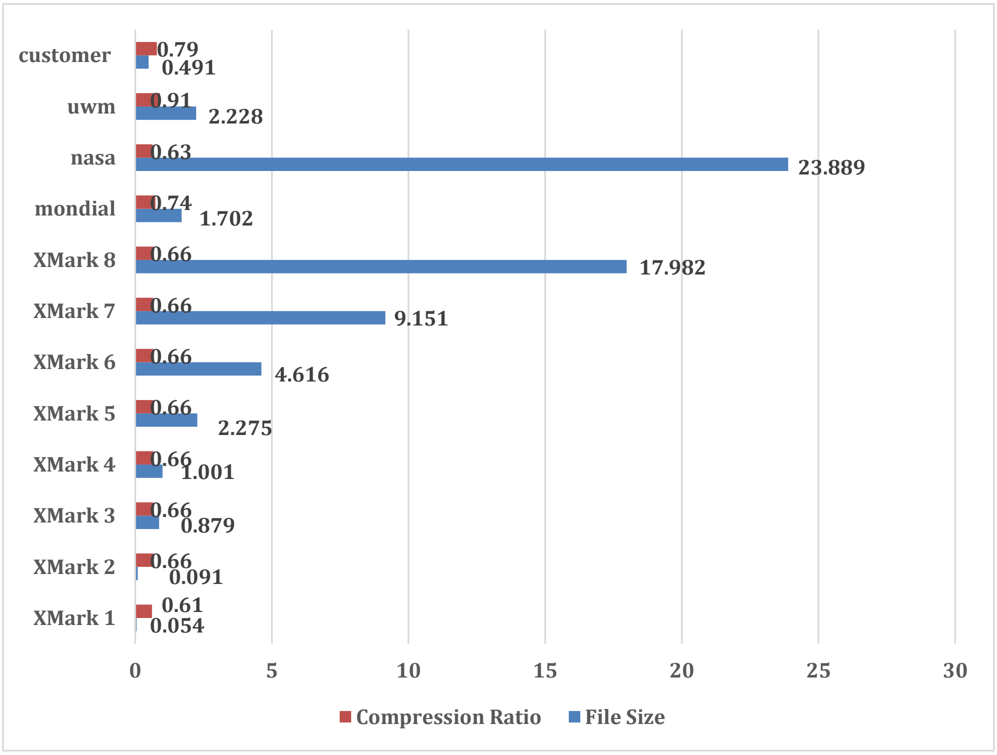


Figure 11.3: Compression Ratio against file size

11.2.1.1 Comparative analysis

SIQXC's compression ratio is compared with compression ratios of other queryable compressors (QC). This is done in two ways; using the average compression ratio and the compression ratios different compressors over the same datasets. The data about other datasets was obtained from one source for consistency [AlHamadani, 2011]. The compressors were tested over the same datasets; Shakespear, Swisspprot, Treebank, LineItem, UW course data, NASA and DBLP [AlHamadani, 2011]. The specific compression ratio are shown in Appendix III. The NASA and uwm files used to test SIQXC have been used to compare SIQXC to some of the existing compressors.

The results obtained from applying SIQXC compression on these files are used to compare this compressor with others. Figure 11.4 shows the performance of SIQXC against the widely used compressors. From the results we observe that SIQXC's compression ratio is higher than most of the compression ratios except that for XSAQCT's which is higher by 0.1. XSAQCT achieves a better compression but support only exact match queries (See Appendix III, for a detailed query support report for the compressors). As mentioned in Chapter 5, a trade off exists between query supports and the compression ratio. Methods that support a wider spectrum of queries usually have a lower compression ratio compared to those that do not and this is demonstrated by these results.

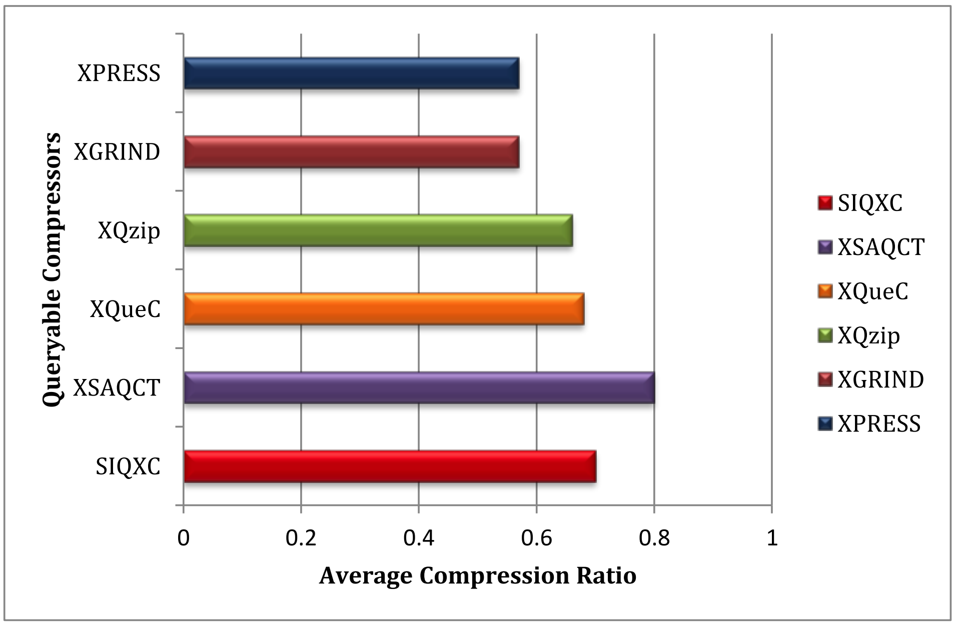


Figure 11.4: Average compression ratios against QC

Figure 11.5 compares SIQXC with non queryable compressors (NQC). Accordingly these compressors should have a better compression ratio than SIQXC but it surpasses them all.

In Figure 11.6 SIQXC's compression ratio of specific datasets is compared to those of other compressors. XSAQCT still has a better compression ratio in compressing the uwm; a highly structured XML file. However, the difference is not that big. XSAQCT achieves a 95% compression whereas SIQXC obtains 91%. SIQXC performs fairly well in comparison with other compressor on the NASA dataset in that it achieves the same compression ratio with XQzip. Its compression ratio is however higher than the other compressors'; XGRIND, XPRESS and XQueC. SIQXC outperforms the rest of the compressors when compressing the highly structured uwm dataset.

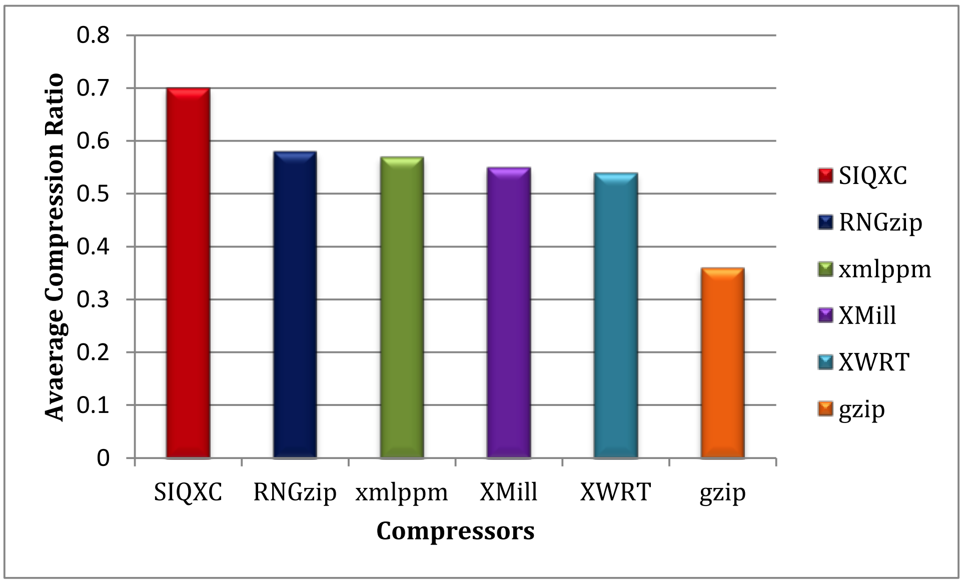


Figure 11.5: Average compression ratio against NQC

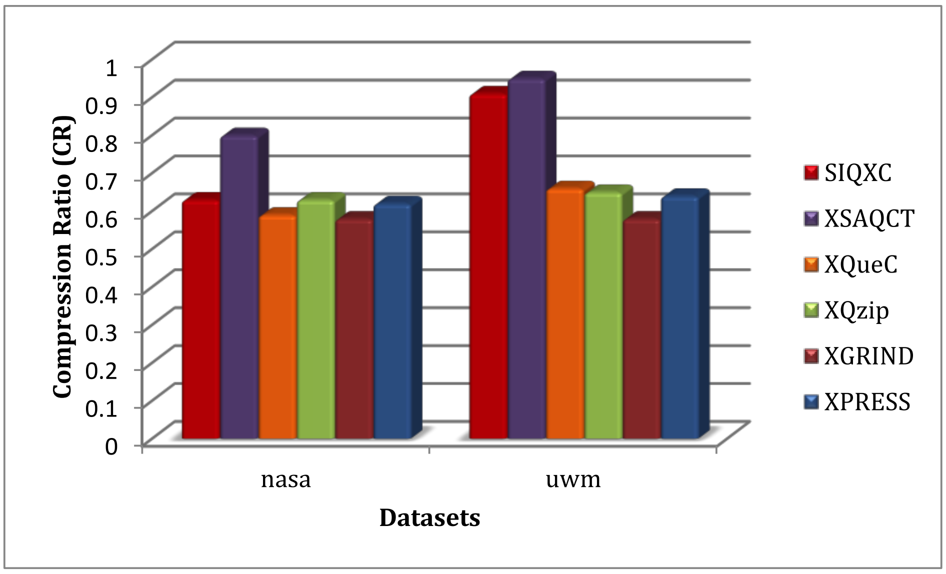


Figure 11.6: Compression ratio in relation to specific datasets

The compression ratio results presented above suggest that SIQXC outperforms other widely used compressor except XSAQCT. It however gains an advantage over XSAQCT but supporting more queries other than just the exact match queries (See Section 11.2.3 for queries supported by SIQXC). It can also observed that it performs better with highly structured documents. SIQXC achieves a poor compression ratio when compressing small files. This behaviour is observed on many compressors even gzip. However, the compression ratio of this compressor is not dependent on the file size. SIQXC achieves a competitive compression ratio and also allows query evaluation on compressed data as shown by the results in Section 11.2.3.

11.2.2 Compression Time

The files used for evaluating the compression ratio of SIQXC were also used to measure compression time. The time is measured in seconds. Recall that in Chapter 10 it was mentioned that the compression time can be expected to increase with the file size. This is clearly shown in Figure 11.7. The results shown do not include the time taken to parse the XML file. This time is excluded because different parsers take different lengths of time to parse an XML file and SIQXC is not restricted to the Document Object Model (DOM) parser only. DOM was only used in this experimental framework for proof of concept. Simple API for XML, SAX, can also be used and this would change the compression time significantly. However, results that include parsing time are also presented in Figure 11.8.

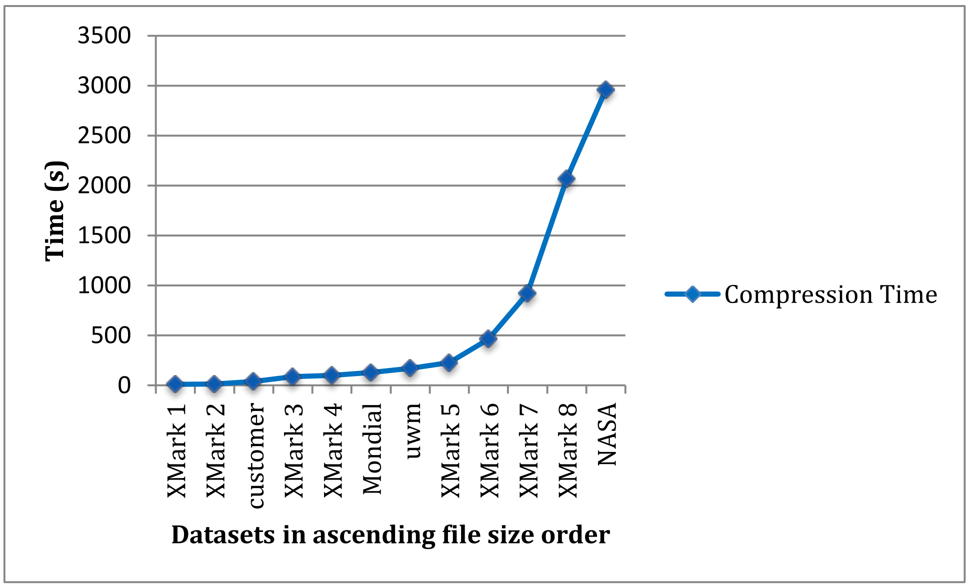


Figure 11.7: Compression time against file size

The results in Figure 11.8 show that parsing significantly increases compression time. In Figure 11.7, the compression time increases with the file size and this behaviour is observed again in Figure 11.8 where parsing time is included. Though a parser affects compression time it does not change the relationship between compression time and the file size.

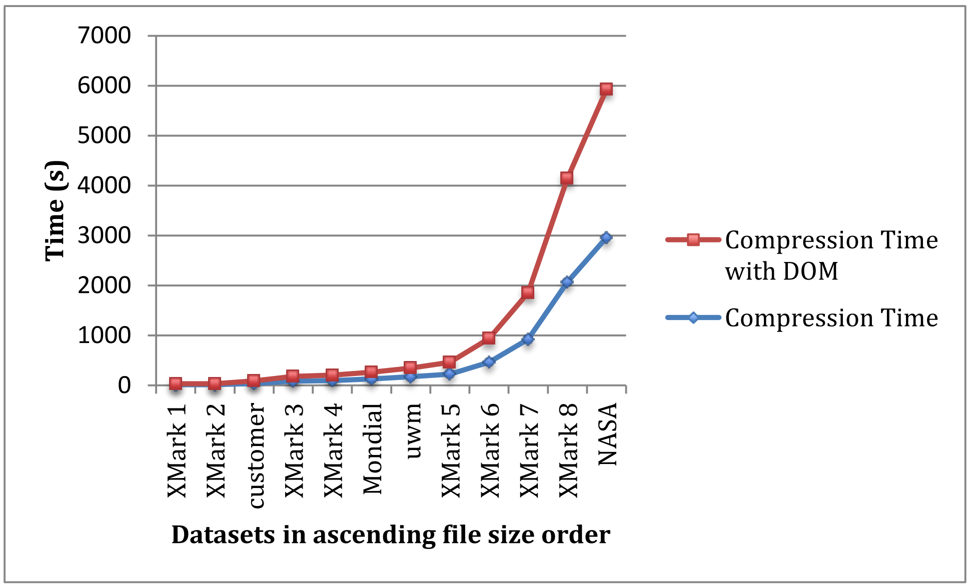


Figure 11.8: Compression time with and without DOM

11.3 The Query Processor

A description of the SIQXC query processor was provided in Chapter 9 outlining the kind of queries it is intended to support and the query processing approach used. According to that discussion SIQXC processes queries based on their type which can be; keyword, simple path, same node predicate, divergent node predicate, and the multiple operator queries. Some of the queries used in testing this query processor are shown in Appendix II. The query analyser analyses the query to determine the type so that the right method can be invoked. The query processor has an analyser and query search engine. In evaluating the query processor each component is tested. The query analyser is tested to see if it can analyse queries correctly. This is because the accuracy of the final result set that the query search engine returns are dependent on the query analyser. Wrong analysis can lead to inaccurate results.

In testing the analyser several queries were run to see which method was invoked. The results indicate that the analyser is precise. This was achieved through observing the execution logs to see which methods were invoked for each run. To establish the average query response time different queries were run on the same file. From these results simple path and keyword queries are much faster to evaluate than the predicate queries. For the predicate queries, same node queries are faster to evaluate than divergent node queries. The multiple operator queries take the longest time than all other queries. Note that at the time of writing this experiment was run on a resource rich environment so the results were not included in this work. Only the keyword and simple path queries were run on a resource constrained environment.

The query response time can be expected to increase with increasing file sizes. This is shown in Figure 11.9 after running the same query on different compressed documents sized document.

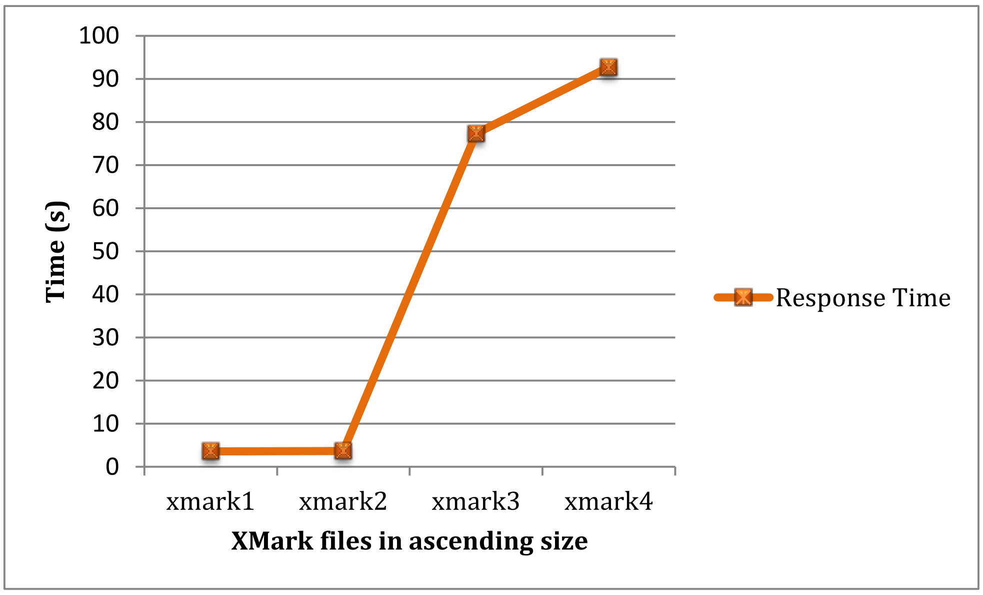


Figure 11.9: Query Response time on different sized XMark files

The results show that the query response time increases with the size of the document. The other observation made during this evaluation was that the query response time is also dependent on the sizes of the containers that are necessary for its evaluation. Generally the response time is dependent on the file size but sometimes the time might be slightly high though the file size is small because the container necessary for query evaluation is relatively big.

From the evaluation, the query analyser is accurate in analysing all types of queries as per the execution logs. The query response time is different for different types of queries. It has been established that excluding the external factors, the query response time is dependent on three factors; file size, container size, query type. Small sized document and containers result in fast response time. Queries that require many containers for evaluation take longer to evaluate than those that need fewer containers. The number of containers required to evaluate a query is dictates the query time. In terms of performance, this work provides preliminary results that form a base for further extensive test. Due to time constraints the performance of the query processor was not extensively tested to provide conclusive results but some relationships that exist thereof have been established.

11.4 Chapter summary

This chapter presented the results obtained from implementing the experimental design in Chapter 10. A set of twelve datasets were used to evaluate the functionality of the compressor and query processor. With the compressor two aspects were measured; the compression ratio and compression time. The compression ratio of SIQXC was compared to other existing queryable compressors by using published results. From the comparative analysis SIQXC achieved a competitive average compression ratio that is highier than that of XGRIND, XPRESS, XQueC, XQzip and all the non queryable methods it was compared to. Its compression ratio on specific datasets was also compared against the compression ratio of other compressors on the same data. On the NASA dataset SIQXC had the same compression ratio as XQzip but outperformed it in compressing the uwm dataset. SIQXC was better than XGRIND, XPRESS and XQueC for both datasets.

The compression time results indicate that compression time increases with the file size. SIQXC supports different types of queries as described in Chapter 9. These queries were run on the files created from the XMark benchmark and the customer file from the Transaction Processing Performance Council (TPC) benchmark.

The results discussed in this Chapter suggests that the process of using the simple 2-tuple integer encoding, clustering and gzip which is derived from combining ideas from labelling schemes and existing compression methods can possibly allow users to locally access a relatively large XML file on their resource restricted smartphones. The next chapter concludes this thesis and suggest some ideas for future work.

CHAPTER 12

CONCLUSION AND FUTURE WORK

12.1 Introduction

This work attempted to develop a schema independent queryable compressor that would to some extent allow users to store and manipulate a relatively large XML file in their resource restricted devices. The research motivation was discussed in Chapter 5 stating the goals and hypothesis. The designs of the compressor and query processor were discussed in Chapter 8 and 9 respectively. Chapter 10 present an experimental framework that described the strategies and tests that were used to evaluate these two components. In that chapter the datasets and benchmarks that are usually used to evaluate XML applications were also reviewed. The results were presented and analysed in Chapter 11. Having analysed and evaluated the results from the experimental framework, this chapter provides the findings highlighting whether the hypothesis has been proven or not. Suggestions on future improvements are also made in this chapter from the limitations discovered during the testing and evaluation of SIQXC.

12.2 Main contributions of this research

The research provided a partial solution to the XML storage space problem on smartphones by exploiting ideas from labelling schemes and compression to design a queryable compressor that has a competitive compression ratio. It has contributed to literature by providing an improved solution to the XML storage problem on resource constrained devices by proposing a novel simple 2 tuple integer encoding system that when combined with clustering significantly removes the redundancy XML. It also provided means of extracting the compressed XML data through a query processor that runs on these devices. The research provided empirical proof of the Schema Independent Queryable XML Compressor by capturing the compression time, compression ratio, query response time and the accuracy of the result set of each query type.

12.3 Relating Research Results to the Hypothesis

With regard to the hypothesis the results of this research suggest that it is possible to combine ideas from labelling schemes and existing compression methods to compress data enough to some extent to give users access to their data locally. Ideas from labelling schemes were used to come up with a novel simple 2 tuple integer encoding system that was used to prune trees. Combining this with clustering and using gzip as a backend compressor achieves a competitive compression ratio thereby allowing users to access the otherwise verbose XML on their smartphones. The hypothesis was tested using eleven datasets to establish compression time and ratio and different query types to evaluate the functionality of the query processor.

12.4 Future work

SIQXC provides storage and access of XML on resource constrained devices through a compressor with a competitive ratio and a query processor that run on these devices. This has been established through an empirical proof however there are improvements that can be made as suggested below.

More query functions can be added to SIQXC like the support for updates. Due to the way data is stored in its compressed state, the use of a log of updates could be used. The log would capture all updates that a user makes and these would then be used to update the actual XML document. Other functions that may be incorporated include positional queries and aggregation however careful considerations must be made because supporting too many functions usually compromises the compression ratio. The query strategy used in SIQXC can also be improved to include more XPATH restrictions.

This thesis provides a design for the decompressor but it was not tested since it is not relevant for this work because this work does not support updates. The decompressor can be implemented for scenarios where a user transfers file between two resource rich environments through a resource constrained device.

Sibling order is lost during compression in SIQXC to improve compression ratio and also because it is usually not that useful in the real world situations. For example, a user hardly asks for a third child of a specific node, they normally use the tag name than sibling order. Sibling order is necessary during transmission and in cases where the XML data is managed on different devices so if a decompressor were to be developed sibling order must be maintained. This can be accomplished through using a 3 tuple integer encoding system that holds the level, sub tree identity and sibling order. Having a label like **3.2.5** would mean that the node is level three (**3**), it belongs to sub tree two (**2**) and it is the fifth child (**5**). The other way would be to use a dictionary that only stores sibling order information. This dictionary would not need to be loaded into the resource constrained device because it is only necessary during decompression.

The current implementation is for the android platform only. Android was chosen because it is the most widely used mobile platform. SIQXC can be extended to run on other platforms like iOS, Windows mobile, Symbian and Blackberry and other future platforms.

CHAPTER 13

REFERENCES

Adiego, J., de la Puente, P. and Navarro, G. (2004). Merging prediction by partial matching with structural contexts model. p.522.

Adiego, J., Navarro, G. and de la Fuente, P. (2003). SCM: Structural contexts model for improving compression in semistructured text databases. pp.153--167.

Adiego, J., Navarro, G. and others, (2007). Lempel-Ziv compression of highly structured documents. *Journal of the American Society for Information Science and Technology*, 58(4), pp.461--478.

Ahn, C., Li, Q., Elmasri, R., Prabhakar, S., Manandhar, N. and Kim, D. (2005). A Survey of Three Types of XML Indexing Techniques. *ACM Transactions on Computational Logic*, 37(4), p.12.

Al-Hamadani, B., Lu, Z. and Alwan, R. (2013). Schema Independent XML Compressor. *Information Retrieval Methods for Multidisciplinary Applications*, p.95.

AlHamadani, B. (2011). *Retrieving Information from Compressed XML Documents According to Vague Queries*. PhD. University of Huddersfield.

Al-Khalifa, S., Jagadish, H., Koudas, N., Patel, J., Srivastava, D. and Wu, Y. (2002). Structural joins: A primitive for efficient XML query pattern matching. pp.141--152.

Arion, A., Bonifati, A., Manolescu, I. and Pugliese, A. (2007). XQueC: A query-conscious compressed XML database. *ACM Transactions on Internet Technology (TOIT)*, 7(2), p.10.

REFERENCES

Arion, A., Bonifati, A., Costa, G., d'Aguanno, S., Manolescu, I. and Pugliese, A. (2004). Efficient query evaluation over compressed XML data. *Springer*, pp.200--218.

Arroyuelo, D., C'anovas, R., Navarro, G. and Sadakane, K. (2010). Succinct Trees in Practice. pp.84--97.

Böhme, T. and Rahm, E. (2001). XMach-1: A benchmark for XML data management. pp.264--273.

Barbosa, D., Manolescu, I. and Yu, J. (2009). XML Benchmarks. *Encyclopedia of Database Systems*, pp.3576--3579.

Bex, G., Neven, F. and Vansummeren, S. (2007). Inferring XML schema definitions from XML data. pp.998--1009.

Bex, G., Neven, F. and Van den Bussche, J. (2004). DTDs versus XML schema: a practical study. pp.79--84.

Boeckmann, B., Bairoch, A., Apweiler, R., Blatter, M., Estreicher, A., Gasteiger, E., Martin, M., Michoud, K., O'Donovan, C., Phan, I. and others, (2003). The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003. *Nucleic acids research*, 31(1), pp.365--370.

Boncz, P., Grust, T., Van Keulen, M., Manegold, S., Rittinger, J. and Teubner, J. (2006). MonetDB/XQuery: a fast XQuery processor powered by a relational engine. pp.479--490.

Bosak, J. (1999). *Shakespeare 2.00*. [Online] Research.cs.wisc.edu. Available at: http://research.cs.wisc.edu/niagara/data/shakes/shaksper.htm [Accessed 29 May 2014].

Boulos, M., Wheeler, S., Tavares, C. and Jones, R. (2011). How smartphones are changing the face of mobile and participatory healthcare: an overview, with example from eCAALYX. *Biomedical engineering online*, 10(1), p.24.

REFERENCES

Brenes Barahona, S. (2011). Structural summaries for efficient XML query processing. *Indiana University*.

Bressan, S., Lee, M., Li, Y., Lacroix, Z. and Nambiar, U. (2003). The XOO7 benchmark. *Springer*, pp.146--147.

Brisaboa, N., Farina, A., Navarro, G. and Esteller, M. (2003). (S, C)-dense coding: An optimized compression code for natural language text databases. pp.122--136.

Brodianskiy, T. and Cohen, S. (2007). Self-correcting queries for xml. pp.11--20.

Bruno, N., Koudas, N. and Srivastava, D. (2002). Holistic twig joins: optimal XML pattern matching. pp.310--321.

Byun, C. and Park, S. (2010). A Schema Based Approach to Valid XML Access Control. *J. Inf. Sci. Eng.*, 26(5), pp.1719--1739.

Carey, M., DeWitt, D. and Naughton, J. (1993). The 007 Benchmark. In: *ACM*. pp.12--21.

Chang, Y., Luo, C. and Huang, C. (2009). Efficient evaluation of XML twig queries with keyword constraints. *Journal of the Chinese Institute of Engineers*, 32(4), pp.469--480.

Chareen, S., Xie, H. and Cole, P. (2008). Energy Efficiency in Mobile Phones: A Survey. *School of Information Technology, Murdoch University*.

Chen, Q., Lim, A. and Ong, K. (2003). D (k)-index: An adaptive structural summary for graph-structured data. pp.134--144.

Chen, S., Li, H., Tatemura, J., Hsiung, W., Agrawal, D. and Candan, K. (2006). Twig 2 Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. pp.283--294.

Chen, T., Lu, J. and Ling, T. (2005). On boosting holism in XML twig pattern matching using structural indexing techniques. pp.455--466.

Chen, Y., Davidson, S. and Zheng, Y. (2006). An efficient XPath query processor for XML streams. pp.79--79.

Cheney, J. (2001). Compressing XML with multiplexed hierarchical PPM models. pp.163--172.

Cheney, J. (2005). An Empirical Evaluation of Simple DTD-Conscious Compression Techniques. pp.43--48.

Cheney, J. (2006). Tradeo_s in XML Database Compression. *Data Compression Conference*.

Cheng, J. and Ng, W. (2004). XQzip: Querying compressed XML using structural indexing. *Springer*, pp.219--236.

Chung, C., Min, J. and Shim, K. (2002). APEX: An adaptive path index for XML data. pp.121--132.

Cleary, J. and Teahan, W. (1997). Unbounded length contexts for PPM. *The Computer Journal*, 40(2 and 3), pp.67--75.

Cleary, J. and Witten, I. (1984). Data compression using adaptive coding and partial string matching. *Communications, IEEE Transactions on*, 32(4), pp.396--402.

Cohen, E., Kaplan, H. and Milo, T. (2002). Labeling dynamic xml trees. pp.271--281.

Cover, R. (2000). *Cover Pages: NASA Goddard Astronomical Data Center (ADC) 'Scientific Dataset' XML*. [Online] Xml.coverpages.org. Available at: http://xml.coverpages.org/nasa-adc.html [Accessed 4 Jun. 2014].

Cuervo, E., Balasubramanian, A., Cho, D., Wolman, A., Saroiu, S., Chandra, R. and Bahl, P. (2010). MAUI: making smartphones last longer with code offload. pp.49--62.

Deutsch, L. (1996). DEFLATE compressed data format specification version 1.3.

Deutsch, L. (1996). GZIP file format specification version 4.3.

REFERENCES

Deutsch, L. (1996). GZIP file format specification version 4.3.

Dhingra, P. and Swanson, T. (2007). *Microsofttextregistered sql server 2005*.

Etheridge, D. (2013). Developing Androidâ„¢ applications for ARMtextregistered Cortex™-A8 cores. *Texas Instruments: Autor. Recuperado Mayo*, 3.

Fei, Y., Zhong, L. and Jha, N. (2008). An energy-aware framework for dynamic software management in mobile computing systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), p.27.

Feldspar, A. (1997). *An Explanation of the `Deflate' Algorithm*. [Online] Zlib.net. Available at: http://zlib.net/feldspar.html [Accessed 17 Jun. 2014].

Ferragina, P., Luccio, F., Manzini, G. and Muthukrishnan, S. (2006). Compressing and searching XML data via two zips. pp.751--760.

Ferragina, P., Luccio, F., Manzini, G. and Muthukrishnan, S. (2005). Structuring labeled trees for optimal succinctness, and beyond. pp.184--193.

Fixtradingcommunity.org, (n.d.). *Home Page - FIX Trading Community*. [Online] Available at: http://www.fixtradingcommunity.org [Accessed 12 Jun. 2014].

Franceschet, M. (2005). XPathMark: an XPath benchmark for the XMark generated data. *Springer*, pp.129--143.

Franceschet, M. (2005). *XPathMark*. [Online] Sole.dimi.uniud.it. Available at: http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/index.html [Accessed 19 Jun. 2014].

Gao, J., Lu, J., Wang, T. and Yang, D. (2010). Efficient evaluation of query rewriting plan over materialized XML view. *Journal of Systems and Software*, 83(6), pp.1029--1038.

Girardot, M. and Sundaresan, N. (2000). < i> Millau</i>: an encoding format for efficient representation and exchange of XML over the Web. *Computer Networks*, 33(1), pp.747--765.

Gire, F. and Idabal, H. (2008). Updates and views dependencies in semi-structured databases. pp.159--168.

Goadrich, M. and Rogers, M. (2011). Smart smartphone development: iOS versus android. pp.607--612.

Goldman, R. and Widom, J. (1997). Dataguides: Enabling query formulation and optimization in semistructured databases. *Stanford*.

Gou, G. and Chirkova, R. (2007). Efficiently querying large XML data repositories: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 19(10), pp.1381--1403.

Gronli, T., Hansen, J. and Ghinea, G. (2010). Android vs Windows Mobile vs Java ME: a comparative study of mobile development environments. p.45.

Grimsmo, N. and Bjorklund, T. (2010). Towards unifying advances in twig join algorithms. pp.57--66.

Gulhane, V. and Ali, M. (2013). Survey over Adaptive Compression Techniques. *International Journal of Engineering Science and Innovative Technology (IJESIT)*, 2(1), pp.152-156.

Härder, T., Haustein, M., Mathis, C. and Wagner, M. (2007). Node labeling schemes for dynamic XML documents reconsidered. *Data & Knowledge Engineering*, 60(1), pp.126--149.

Hariharan, S. and Shankar, P. (2005). Compressing XML documents with finite state automata.

Haw, S. and Lee, C. (2007). Structural Query Optimization in Native XML Databases: A Hybrid Approach. *Journal of Applied Sciences*, 7(20), pp.2934--2946.

Haw, S. and Lee, C. (2008). TwigINLAB: A decomposition-matching-merging approach to improving XML query processing. *American Journal of Applied Sciences*, 5(9), p.1199.

Haw, S. and Lee, C. (2009). Extending path summary and region encoding for efficient structural query processing in native XML databases. *Journal of Systems and Software*, 82(6), pp.1025--1035.

Haw, S. and Lee, C. (2008). Evolution of structural path indexing techniques in XML databases: A survey and open discussion. pp.2054--2059.

Haw, S. and Rao, G. (2007). Path Query Processing in Large-Scale XML Databases. *Journal of Applied Sciences*, 7(19), pp.2736--2743.

Haw, S. and Lee, C. (2007). INLAB: Improving XML Path Query Optimization. *Journal of Applied Computer Science*, 15(1), pp.47--61.

Haw, S. and Lee, C. (2008). TwigX-Guide: twig query pattern matching for XML trees. *American Journal of Applied Sciences*, 5(9), p.1212.

He, H., Wang, H., Yang, J. and Yu, P. (2005). Compact reachability labeling for graph-structured data. pp.594--601.

Hoque, N., Araki, S., Umeno, H. and Aoyama, T. (2007). Designing an XMLDB for an Embedded System. pp.1265--1269.

Hruvska, P., Martinovic, J., Dvorsk`y, J. and Sn'avsel, V. (2010). XML Compression Improvements Based on the Clustering of Elements. *CISIM, Poland*.

Hu, W., Chen, T., Shi, Q. and Lou, X. (2010). Smartphone software development course design based on android. pp.2180--2184.

Jedidi, A., Arfaoui, O. and Sassi-Hidri, M. (2012). Indexing compressed XML documents. *Springer*, pp.319--328.

Jiang, H., Lu, H. and Wang, W. (2004). Efficient processing of XML twig queries with OR-predicates. pp.59--70.

Jiang, J., Chen, K., Li, X., Chen, G. and Shou, L. (2009). Efficient processing of ordered XML twig pattern matching based on extended Dewey. *Journal of Zhejiang University SCIENCE A*, 10(12), pp.1769--1783.

REFERENCES

Karlsson, J., Lal, A., Leung, C. and Pham, T. (2001). IBM DB2 everyplace: A small footprint relational database system. pp.0230--0230.

Kaushik, R., Bohannon, P., Naughton, J. and Korth, H. (2002). Covering indexes for branching path queries. pp.133--144.

Khaing, A. and Thein, N. (2006). A persistent labeling scheme for dynamic ordered XML trees. pp.498--501.

Khoussainov, B. and Khoussainova, N. (2014). Deterministic Finite Automata.

Kieffer, J., Yang, E., Nelson, G. and Cosman, P. (2000). Universal lossless compression via multilevel pattern matching. *Information Theory, IEEE Transactions on*, 46(4), pp.1227--1245.

Kim, H., Agrawal, N. and Ungureanu, C. (2012). Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4), p.14.

Knuth, D. (1985). Dynamic huffman coding. *Journal of algorithms*, 6(2), pp.163--180.

Krashinsky, R. and Balakrishnan, H. (2005). Minimizing energy for wireless web access with bounded slowdown. *Wireless Networks*, 11(1-2), pp.135--148.

League, C. and Eng, K. (2007). Schema-based compression of XML data with relax NG. *Journal of Computers*, 2(10), pp.9--17.

Leighton, G., Diamond, J. and Muldner, T. (2005). AXECHOP: a grammar-based compressor for XML. p.467.

Leighton, G., MÃ¼ldner, T. and Diamond, J. (2005). TREECHOP: A Tree-based Query-able Compressor for XML.

Ley, M., Herbstritt, M., Ackermann, M., Wagner, M., Hoffmann, O., Schwarz, R. and Keutz, S. (2005). *Index of /xml*. [Online] DBLP. Available at: http://dblp.uni-trier.de/xml/ [Accessed 15 Jun. 2014].

Li, C. and Ling, T. (2005). An improved prefix labeling scheme: A binary string approach for dynamic ordered XML. pp.125--137.

Li, J. and Wang, J. (2008). TwigBuffer: avoiding useless intermediate solutions completely in twig joins. pp.554--561.

Li, W. (2003). Xcomp: An XML compression tool.

Li, Y., Bressan, S., Dobbie, G., Lacroix, Z., Lee, M., Nambiar, U. and Wadhwa, B. (2001). XOO7: applying OO7 benchmark to XML query processing tool. pp.167--174.

Li, Y. (n.d.). XML BENCHMARKS PUT TO THE TEST.

Lian, W., Mamoulis, N., Cheung, D. and Yiu, S. (2005). Indexing useful structural patterns for XML query processing. *Knowledge and Data Engineering, IEEE Transactions on*, 17(7), pp.997--1009.

Liao, I., Hsu, W. and Chen, Y. (2010). An efficient indexing and compressing scheme for XML query processing. *Springer*, pp.70--84.

Liefke, H. and Suciu, D. (2000). XMill: an efficient compressor for XML data. 29(2), pp.153--164.

Lin, Y., Zhang, Y., Li, Q. and Yang, J. (2005). Supporting efficient query processing on compressed XML files. pp.660--665.

Liu, J. and Roantree, M. (2010). OTwig: An Optimised Twig Pattern Matching Approach for XML Databases. *Springer*, pp.564--575.

Liu, J., Roantree, M. and Bellahsene, Z. (2010). A schema guide for accelerating the view adaptation process. *Springer*, pp.160--173.

Liu, M. and Ling, T. (2002). Towards declarative XML querying. pp.127--136.

Lo, A., "Ozyer, T., Tahboob, R., Kianmehr, K., Jida, J. and Alhajj, R. (2010). XML materialized views and schema evolution in VIREX. *Information Sciences*, 180(24), pp.4940--4957.

Lu, E. and Cheng, Y. (2004). Design and implementation of a mobile database for Java phones. *Computer Standards & Interfaces*, 26(5), pp.401--410.

Lu, J., Chen, T. and Ling, T. (2004). Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. pp.533--542.

Lu, J., Chen, T. and Ling, T. (2005). TJFast: effective processing of XML twig pattern matching. pp.1118--1119.

Lu, J., Ling, T., Chan, C. and Chen, T. (2005). From region encoding to extended dewey: On efficient processing of XML twig pattern matching. pp.193--204.

Lu, J., Ling, T., Bao, Z. and Wang, C. (2011). Extended xml tree pattern matching: theories and algorithms. *Knowledge and Data Engineering, IEEE Transactions on*, 23(3), pp.402--416.

Müldner, T., Fry, C., Miziolek, J. and Durno, S. (2009). Xsaqct: Xml queryable compressor. pp.11--14.

Madden, S., Franklin, M., Hellerstein, J. and Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1), pp.122--173.

Madria, S., Chen, Y., Passi, K. and Bhowmick, S. (2007). Efficient processing of XPath queries using indexes. *Information Systems*, 32(1), pp.131--159.

Manandhar, N. (2007). Structure Based XML Indexing. *Computer Science & Engineering*.

Mandhani, B. and Suciu, D. (2005). Query caching and view selection for XML databases. pp.469--480.

Marcus, M., Marcinkiewicz, M. and Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2), pp.313--330.

Meng, X., Wang, Y., Luo, D., Lu, S., An, J., Chen, Y., Ou, J. and Jiang, Y. (2003). OrientX: A Schema-based Native XML Database System.

Milo, T. and Suciu, D. (1999). Index Structures for Path Expressions. In: *Springer-Verlag*. pp.277--295.

Min, J., Park, M. and Chung, C. (2003). XPRESS: A queriable compression for XML data. pp.122--133.

Mlýnková, I. and Nečaský, M. (2009). Towards inference of more realistic XSDs. pp.639--646.

Mohan, P., Padmanabhan, V. and Ramjee, R. (2008). Nericell: rich monitoring of road and traffic conditions using mobile smartphones. pp.323--336.

Moro, M., Vagena, Z. and Tsotras, V. (2005). Tree-pattern queries on a lightweight XML processor. pp.205--216.

Nair, S. (n.d.). [Online] Available at: https://people.ok.ubc.ca/rlawrenc/research/Students/SN_04_XMLCompress.pdf [Accessed 17 Dec. 2013].

Ng, W., Lam, W., Wood, P. and Levene, M. (2006). XCQ: A queriable XML compression system. *Knowledge and Information Systems*, 10(4), pp.421--452.

Nicola, M., Kogan, I. and Schiefer, B. (2007). An XML transaction processing benchmark. pp.937--948.

Nicola, M. and Van der Linden, B. (2005). Native XML support in DB2 universal database. pp.1164--1174.

O'Connor, M. and Roantree, M. (2010). Desirable properties for XML update mechanisms. p.23.

Oliver, E. (2009). A survey of platforms for mobile networks research. *ACM SIGMOBILE Mobile Computing and Communications Review*, 12(4), pp.56--63.

O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G. and Westbury, N. (2004). ORDPATHs: insert-friendly XML node labels. pp.903--908.

Oracle, (2006). *Oracle Database Lite Client 10g*. [Online] Oracle.com. Available at: http://www.oracle.com/technetwork/database/database-lite/lite-client-090611.html [Accessed 6 Apr. 2014].

Ortiz, S. (2000). Embedded databases come out of hiding. *Computer*, 33(3), pp.16--19.

Pavlov, I. (n.d.). *7-Zip*. [Online] 7-zip.org. Available at: http://www.7-zip.org/ [Accessed 6 Mar. 2014].

Pena, A. (2013). Compressed self-indexed xml representation with efficient xpath evaluation. PhD. Universidade da Coruña.

Phillips, D., Zhang, N., Ilyas, I. and Ozsu, M. (2006). InterJoin: Exploiting indexes and materialized views in XPath evaluation. pp.13--22.

Portokalidis, G., Homburg, P., Anagnostakis, K. and Bos, H. (2010). Paranoid Android: versatile protection for smartphones. pp.347--356.

Pucheral, P., Bouganim, L., Valduriez, P. and Bobineau, C. (2001). PicoDBMS: Scaling down database techniques for the smartcard. *The VLDB Journal*, 10(2-3), pp.120--132.

Purwitasari, D., Purwananto, Y. and Prasetyo, A. (n.d.). Compression on XML data with reverse arithmetic encoding: A case study.

Qiao, D. and Shin, K. (2005). Smart power-saving mode for IEEE 802.11 wireless LANs. pp.1573-15833.

Qin, L., Yu, J. and Ding, B. (2007). TwigList: make twig pattern matching fast. *Springer*, pp.850--862.

Rahmati, A. and Zhong, L. (2007). Context-for-wireless: context-sensitive energy-efficient wireless data transfer. pp.165--178.

Rizzolo, F. (2001). ToXin: an indexing scheme for XML data. PhD. University of Toronto

Robie, J. (2007). XML processing and data integration with XQuery. *Internet Computing, IEEE*, 11(4), pp.62--67.

Rocco, D., Caverlee, J. and Liu, L. (2005). XPACK: A High-Performance WEB Document Encoding. In: *INSTICC Press*. pp.32-39.

Runapongsa, K., Patel, J., Jagadish, H., Chen, Y. and Al-Khalifa, S. (2006). The Michigan benchmark: towards XML query performance diagnostics. *Information Systems*, 31(2), pp.73--97.

Ryabko, B. and Rissanen, J. (2003). Fast adaptive arithmetic code for large alphabet sources with asymmetrical distributions. *Communications Letters, IEEE*, 7(1), pp.33--35.

Sakr, S. (2009). XML compression techniques: A survey and comparison. *Journal of Computer and System Sciences*, 75(5), pp.303--322.

Salomon, D. (2004). *Data Compression: The Complete Reference*. ed. p.899.

Satyanarayanan, M., Lewis, G., Morris, E., Simanta, S., Boleng, J. and Ha, K. (2013). The role of cloudlets in hostile environments. *IEEE Pervasive Computing*, 12(4), pp.0040--49.

Schlieder, T. (2002). Schema-driven evaluation of approximate tree-pattern queries. *Springer*, pp.514--532.

Schmidt, A., Waas, F., Kersten, M., Carey, M., Manolescu, I. and Busse, R. (2002). XMark: A benchmark for XML data management. pp.974--985.

Schmidt, A. (2001). *XMark - An XML BenchmarkProject*. [Online] Xml-benchmark.org. Available at: http://www.xml-benchmark.org/ [Accessed 28 May 2014].

Scioscia, F. and Tinelli, E. (2011). A Framework for Query Processing over Compressed Knowledge Bases. 2, pp.86--91.

Segoufin, L. (2003). Typing and querying XML documents: some complexity bounds. pp.167--178.

Seltzer, M. and Oracle, (2007). *Guide to Oracle Berkeley DB for SQL Developers*. [Online] Oracle.com. Available at: http://www.oracle.com/technetwork/articles/seltzer-berkeleydb-sql-086752.html [Accessed 19 Feb. 2014].

REFERENCES

Senthilkumar, R. and Arputharaj, K. (2011). Efficiently Querying the Indexed Compressed XML Data (IQX). *Journal of Database Management System (IJDMS)*.

Sharma, K. (2011). Android in opposition to iPhone. *International Journal on Computer Science and Engineering*, 3(5), pp.1965--1969.

Shirani, S. (n.d.). Data Compression: The Complete Reference by D. Salomon, Springer, 2007, ISBN-13: 978-1-84628-602-5, 1,092 pages, hardbound. Reviewed.

Skibiński, P. and Swacha, J. (2007). Combining efficient XML compression with query processing. pp.330--342.

Su-Cheng, H., Chien-Sing, L. and others, (2009). Efficient Preprocesses for Fast Storage and Query Retrieval in Native XML Database. *IETE Technical Review*, 26(1), p.28.

Su-Cheng, H., Chien-Sing, L. and others, (2009). Node labeling schemes in XML query optimization: a survey and trends. *IETE Technical Review*, 26(2), p.88.

Suciu, D. (1992). Treebank: XML data repository. *Rapport technique, University of Pennsylvania Treebank Project, Novmber*.

Suciu, D. and Miklau, G. (n.d.). *UW XML Repository*. [Online] Cs.washington.edu. Available at: http://www.cs.washington.edu/research/xmldatasets/ [Accessed 1 Jun. 2014].

Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E. and Zhang, C. (2002). Storing and querying ordered XML using a relational database system. pp.204--215.

Teahan, W. and Cleary, J. (1996). The entropy of English using PPM-based models. pp.53--62.

Tidwell, D. (2002). [Online] Available at: http://www.ibm.com/developerworks/xml/tutorials/xmlintro/section2.html [Accessed 4 Jul. 2011].

Tolani, P. and Haritsa, J. (2002). XGRIND: A query-friendly XML compressor. pp.225--234.

Toman, V. and others, (2004). Syntactical compression of XML data.

Tpc.org, (2001). *TPC-H - Homepage*. [Online] Available at: http://www.tpc.org/tpch/ [Accessed 1 Jun. 2014].

Verbelen, T., Simoens, P., De Turck, F. and Dhoedt, B. (2013). Leveraging cloudlets for immersive collaborative applications. *IEEE Pervasive Computing*, 12(4), pp.30--38.

Vidal, V. and Casanova, M. (2003). Efficient maintenance of xml views using view correspondence assertions. *Springer*, pp.281--291.

W3c, (2010). *XML Essentials - W3C*. [Online] W3.org. Available at: http://www.w3.org/standards/xml/core [Accessed 19 May 2011].

W3C, (n.d.). *Efficient XML Interchange Working Group Public Page*. [Online] W3.org. Available at: http://www.w3.org/XML/EXI/ [Accessed 7 Mar. 2014].

Wang, H., Li, J., Luo, J. and He, Z. (2004). XCpaqs: Compression of XML document with XPath query support. 1, pp.354--358.

Wang, H., Li, J., Liu, X. and Luo, J. (2009). Query Optimization for Complex Path Queries on XML Data. pp.389--404.

Wang, H., Park, S., Fan, W. and Yu, P. (2003). ViST: a dynamic index method for querying XML data by tree structures. pp.110--121.

Wei, D. and Wei, X. (2012). Structural join oriented xml data compression. pp.29--33.

Weigel, F., Schulz, K. and Meuss, H. (2005). The bird numbering scheme for xml and tree databases--deciding and reconstructing tree relations using efficient arithmetic operations. *Springer*, pp.49--67.

Weimin, W., Huijiang, G., Yi, H., Jingbao, F. and Huan, W. (2008). Improvable deflate algorithm. pp.1572--1574.

Weiner, A., Mathis, C. and H"arder, T. (2008). Towards cost-based query optimization in native xml database management systems. *Citeseer*.

Whang, K. (2007). A New DBMS Architecture for DB-IR Integration. In: *Springer-Verlag*. pp.4--5.

Whang, K., Lee, M., Lee, J., Kim, M. and Han, W. (2005). Odysseus: A High-Performance ORDBMS Tightly-Coupled with IR Features. In: *IEEE Computer Society*. pp.1104--1005.

Williams, R. (1991). An extremely fast Ziv-Lempel data compression algorithm. pp.362--371.

Witten, I., Neal, R. and Cleary, J. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6), pp.520--540.

Wong, R., Lam, F. and Shui, W. (2007). Querying and maintaining a compact XML storage. pp.1073--1082.

Wu, X., Lee, M. and Hsu, W. (2004). A prime number labeling scheme for dynamic ordered XML trees. pp.66--78.

Xin, Y., He, Z. and Cao, J. (2010). Effective pruning for XML structural match queries. *Data & Knowledge Engineering*, 69(6), pp.640--659.

Xu, X., Feng, Y. and Wang, F. (2009). Efficient processing of XML twig queries with all predicates. pp.457--462.

Yang, C., Chu, Y. and Tsaur, S. (2010). Implementation of a medical information service on Android mobile devices. pp.72--77.

Yao, B., Ozsu, M. and Khandelwal, N. (2004). XBench benchmark and performance testing of XML DBMSs. pp.621--632.

Yu, T., Ling, T. and Lu, J. (2006). TwigStackList$neg$: a holistic twig join algorithm for twig query with not-predicates on XML data. pp.249--263.

Yun, J. and Chung, C. (2008). Dynamic interval-based labeling scheme for efficient XML query and update processing. *Journal of Systems and Software*, 81(1), pp.56--70.

Zhang, C., Naughton, J., DeWitt, D., Luo, Q. and Lohman, G. (2001). On supporting containment queries in relational database management systems. pp.425--436.

Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R., Mao, Z. and Yang, L. (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. pp.105--114.

Zhang, N. and Ozsu, M. (2010). XML native storage and query processing. *Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies. IGI Global*, 699.

Zhang, N., "Ozsu, M., Ilyas, I. and Aboulnaga, A. (2006). Fix: Feature-based indexing technique for XML documents. pp.259--270.

Zhou, J., Xie, M. and Meng, X. (2007). TwigStack+: Holistic twig join pruning using extended solution extension. *Wuhan University Journal of Natural Sciences*, 12(5), pp.855--860.

Zhou, J., Meng, X. and Ling, T. (2009). Efficient processing of partially specified twig pattern queries. *Science in China Series F: Information Sciences*, 52(10), pp.1830--1847.

Zhou, R. (2010). Answering XPath Queries Using XPath Views. *Message from General Chair*, p.37.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3), pp.337--343.

Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5), pp.530--536.

APPENDIX 1

FULL RESULTS FOR THE COMPRESSOR

The Compressor Experiments

Compression Ratio

Compression Ratio (CR)						
File Name	File Size	Size without gzip	size with gzip	CR without gzip	CR with gzip	CR %
XMark 1	0.054	0.025	0.021	0.54	0.61	61
XMark 2	0.091	0.038	0.031	0.58	0.66	66
XMark 3	0.879	0.388	0.301	0.56	0.66	66
XMark 4	1.001	0.447	0.344	0.55	0.66	66
XMark 5	2.275	1.014	0.779	0.55	0.66	66
XMark 6	4.616	2.077	1.558	0.55	0.66	66
XMark 7	9.151	4.124	3.113	0.55	0.66	66
XMark 8	17.982	8.123	6.124	0.55	0.66	66
mondial	1.702	1.11	0.436	0.35	0.74	74
nasa	23.889	12.499	8.827	0.48	0.63	63
uwm	2.228	1.063	0.193	0.52	0.91	91
customer	0.491	0.335	0.105	0.32	0.79	79
			AVERAGE	0.508333333	0.6916667	

Compression Time

COMPRESSION TIME (S)						
File Name	File Size	1st RUN	2nd RUN	3rd RUN	4th RUN	5th Run
XMark 1	0.054	35.982	16.937	13.031	9.005	9.531
XMark 2	0.091	12.416	12.354	11.559	11.845	10.286
customer	0.491	43.827	43.329	40.357	39.658	38.546
XMark 3	0.879	94.603	96.766	87.784	84.31	84.743
XMark 4	1.001	107.607	97.154	99.383	95.23	98.032
mondial	1.702	122.353	122.991	152.65	110.147	117.461
uwm	2.228	177.887	177.999	182.846	164.27	165.234
XMark 5	2.275	262.163	231.002	223.377	228.15	227.408
XMark 6	4.616	521.094	452.845	439.736	431.68	516.534
XMark 7	9.151	1022.4	1037.452	962.898	876.692	922.974
XMark 8	17.982	2187.454	2092.161	2231.309	2017.446	1955.614
nasa	23.889	3179.273	3395.959	2927.766	3017.108	2932.344

CHAPTER 15

APPENDIX II

THE QUERY PROCESSOR

Queries

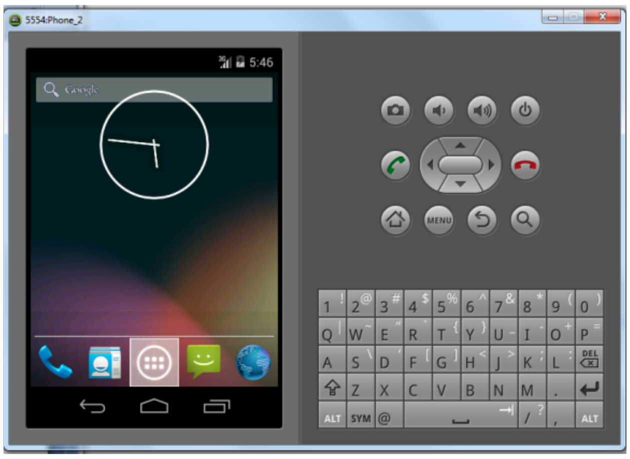
This shows a subset ofspecific queries that were used in testing the query processor. Q1 to Q13 were run on a customer.xml file and Q14 to Q20 were executed on the XMark.xml file

Query Name	**Query**	**Query Type**	**Description**
Q1	C_NAME	Keyword	Exact match keyword query
Q2	/Table/T/C_NAME	Simple Path	Exact math simple path query
Q3	/Table/T [C_CUSTKEY<10]/ C_CUSTKEY	Same Node Predicate	Same node predicate with a single operator
Q4	/Table/T [C_CUSTKEY<10]/C_NAM E	Divergent Node Predicate	Divergent node predicate with a single comparative operator
Q5	/Table/T[C_ACCTBAL>10 000]/ C_ADDRESS	Divergent Node Predicate	Divergent node predicate with a single operator testing a different comparative operator
Q6	/Table/T[C_ACCTBAL=23 79.91]/ C_COMMENT	Divergent Node Predicate	Divergent node predicate with a single operator testing a different comparative operator

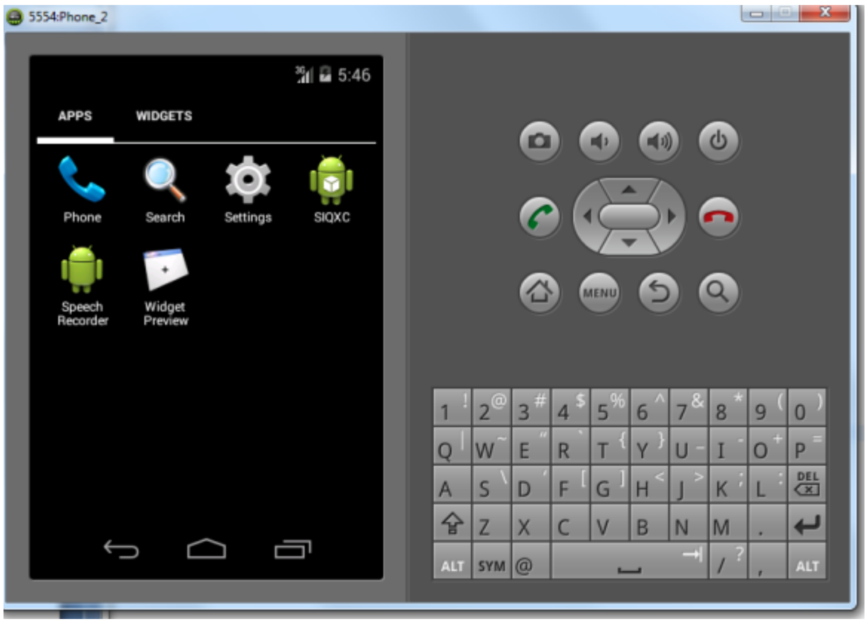
Q7	/Table/T[C_MKTSEGMENT!= AUTOMOBILE]/C_MKTSEGMENT	Same Node Predicate	Same node predicate with a single operator testing a different comparative operator
Q8	/Table/T [C_CUSTKEY>10 AND C_ACCTBAL<1000]/C_NAME	Multiple operator	Conjunctive multiple operator divergent query
Q9	/Table/T [C_NATIONKEY =3 OR C_ACCTBAL>1500]/C_NAME	Multiple operator	Disjunctive multiple operator divergent query
Q10	/Table/T[C_MKTSEGMENT = HOUSEHOLD OR C_NATIONKEY =7]/ C_PHONE	Multiple operator	Disjunctive multiple operator divergent query with more operators
Q11	/Table [C_CUSTKEY<10]/C_NAME	Divergent Node Predicate	Divergent node predicate with wrong path
Q12	T[C_MKTSEGMENT = HOUSEHOLD OR C_NATIONKEY =7]/ C_PHONE	Divergent Node Predicate	Disjunctive multiple operator divergent query with more operators not starting from the root
Q13	Sales	Keyword	Exact match keyword false query
Q14	/site/regions/Africa/item/location	Simple Path	Exact match simple path
Q15	/site/regions/Africa/item[@id='1']/payment	Divergent Node Predicate	Divergent node predicate on an attribute
Q16	/site/regions/Africa/ C_CUSTKEY	Simple Path	Exact match simple path false query
Q17	/site/regions/Africa/item[location!= Liberia]/ @id	Divergent Node Predicate	Divergent node predicate on an attribute with a single comparative operator

Q18	/site/regions/Africa/item/mailboxmail//from	Simple Path	Exact match simple path
Q19	text	Keyword	Exact match keyword query
Q20	/site/regions/Africa/item/mailbox/mail[to=Youjian Siochi mailto:Siochi@uwaterloo.ca]/date	Divergent Node Predicate	Divergent node predicate on a more nested query.

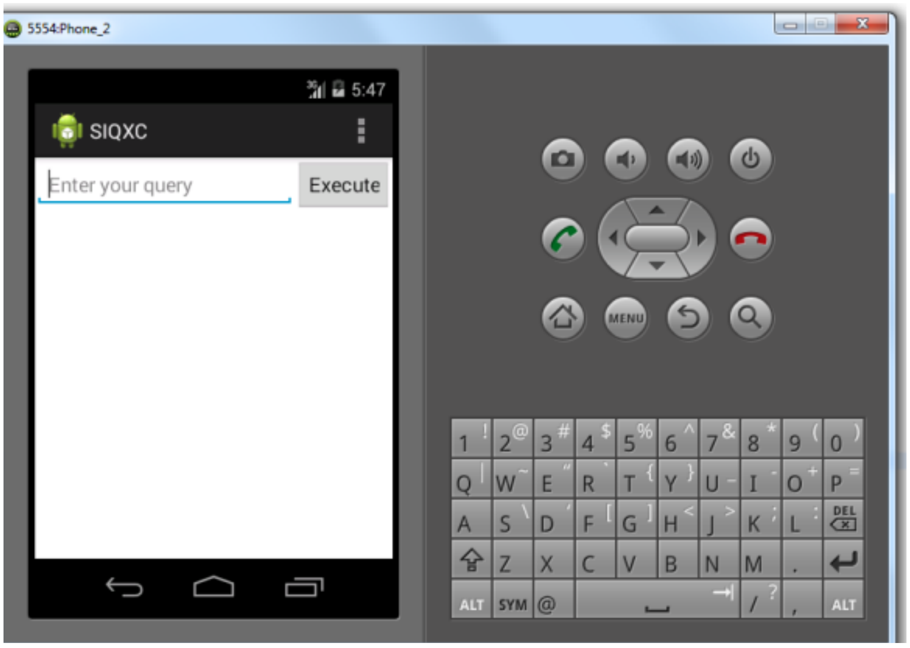
The virtual device used



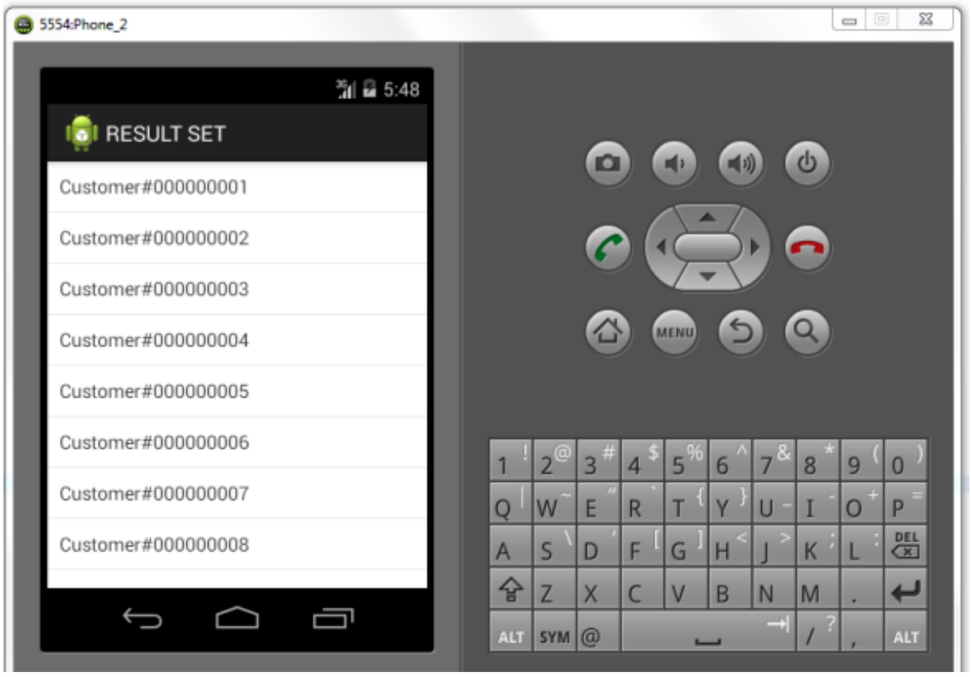
SIQXC installed as an application



The First SIQXC Screen

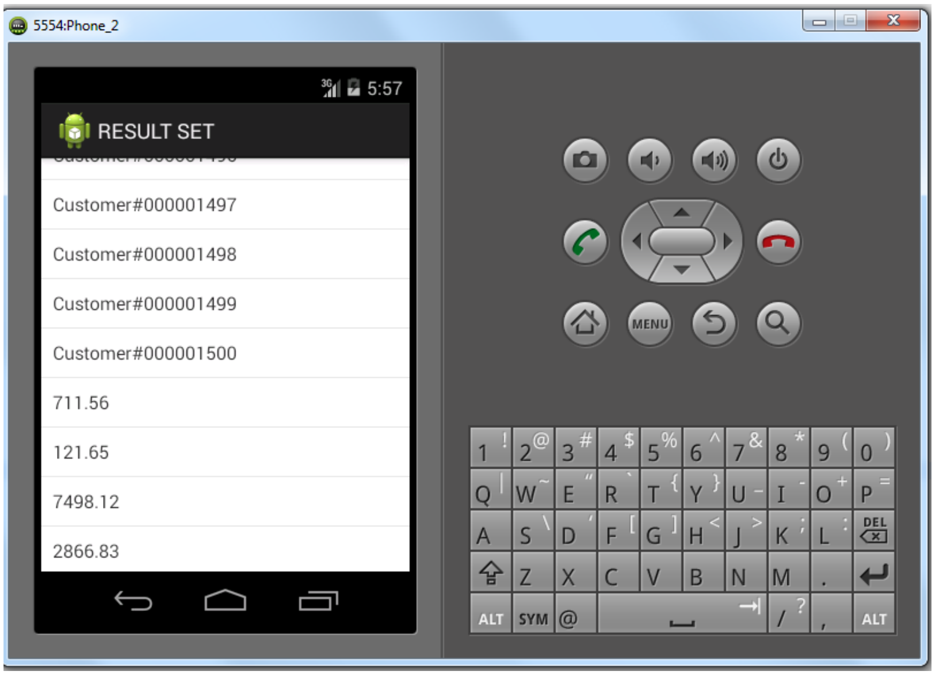


Results from running a keyword query on customer.xml



Customer#000000001
Customer#000000002
Customer#000000003
Customer#000000004
Customer#000000005
Customer#000000006
Customer#000000007
Customer#000000008

Results of a disjunction query with (|) operator



Customer#000001497
Customer#000001498
Customer#000001499
Customer#000001500
711.56
121.65
7498.12
2866.83

CHAPTER 16

APPENDIX III

COMPARISON OF THE COMPRESSORS

Query support

COMPRESSOR	QUERY TYPE				
	Non Predicate		Predicate		
	Keyword	Simple Path	Divergent	Same node	Multiple Operator
SIQXC	•	•	•	•	•
XSAQCT	•	•			
XQueC	•	•	•	•	•
XQzip	•	•	•	•	•
XGRIND	•	•	•	•	•
XPRESS	•	•	•	•	•

Average compression ratio

Compressor	Average Compression Ratio
SIQXC	0.7
XSAQCT	0.8
XQueC	0.68
XQzip	0.66
XGRIND	0.57
XPRESS	0.57