View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Huddersfield Repository

University of HUDDERSFIELD

University of Huddersfield Repository

AlHamadani, Baydaa

Retrieving Information from Compressed XML Documents According to Vague Queries

Original Citation

AlHamadani, Baydaa (2011) Retrieving Information from Compressed XML Documents According to Vague Queries. Doctoral thesis, University of Huddersfield.

This version is available at http://eprints.hud.ac.uk/11179/

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

http://eprints.hud.ac.uk/

Retrieving Information from Compressed XML Documents According to Vague Queries

Baydaa Al-Hamadani

A Thesis Submitted to the University of Huddersfield in Partial Fulfilment of the Requirements for the Degree of Doctor of Philosophy

July, 2011

Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Joan Lu, for her guides and for her continuous support through the different stages of this research. Dr. Joan was always finds the time for me not only to advise me but to listen even to the smallest problems and does her best to help me solving them.

My grateful and thanks go to the University of Huddersfield, who supported me financially and morally and gave me a great opportunity to complete my degree. Continuous thanks go to every member in this great University specially the technician team in the School of Computing and Engineering who never been hesitated or delayed in helping me and all the other researchers.

Great appreciations go to Dr. Christopher Newman and Dr. Hugh Osborn for being very helpful and for their great support to finish this research. I am really grateful for everything they have done to me.

Uncountable thanks to my husband (Raad) for encouraging and motivating me to complete this research. Without him this research would never see the light. I am really grateful for whatever he did to support me and being very patient. Moreover, I would like to thank my son (AlHasan) and my daughter (AlZahraa) for being so patient with me and being very understandable for my situation.

My warm appreciations go to my parents, who spent their lives dreaming of seeing this work completed and who supported me with all they have. I wish I can return a very small part of their favour. I also thank my mother in law, my sisters in law and all my relatives who were praying to see this work in this stage and who were supporting me all over the last few years.

Through the few past years lots of people were supporting and encouraging me. Thank you to all of my friends and colleagues.

Dedication

To my only love, Raad...

To my wonderful parents...

To my son and daughter...

To the unforgettable home...

May Allah who hold the world in his hand, holds them all in his palm.

Abstract

XML has become the standard way for representing and transforming data over the World Wide Web. The problem with XML documents is that they have a very high ratio of redundancy, which makes these documents demanding large storage capacity and high network band-width for transmission. Because of their extensive use, XML documents could be retrieved according to vague queries by naive users with poor background in writing XPath query. The aim of this thesis is to present the design of a system named "XML Compressing and Vague Querying (XCVQ)" which has the ability of compressing the XML document and retrieving the required information from the compressed version with less decompression required according to vague queries.

XCVQ first compressed the XML document by separating its data into containers and then compress these containers using the GZip compressor. The compressed file could be retrieved if a vague query is submitted without the need to decompress the whole file. For the purpose of processing the vague queries, XCVQ decomposes the query according to the relevant documents and then a second decomposition stage is made according to the relevant containers. Only the required information is decompressed and submitted to the user.

To the best of our knowledge, XCVQ is the first XML compressor that has the ability to process vague queries. The average compression ratio of the designed compressor is around 78% which may be considered competitive compared to other queriable XML compressors. Based on several experiments, the query processor part had the ability to answer different kinds of vague queries ranging from simple exact match queries to complex ones that require retrieving information from several compressed XML documents.

Table of **Contents**

Acknowledgments...2
Dedication...3
Abstract...4
Table of **Contents**...5
Copyright Statement...7
List of Tables...8
List of Figures...9
List of Abbreviations...10
CHAPTER 1 Introduction...11
1.1 INTRODUCTION...11
1.2 RESEARCH HYPOTHESIS AND RESEARCH METHODOLOGY...12
1.3 RESEARCH QUESTIONS...14
1.4 MOTIVATIONS AND OBJECTIVES...15
1.5 RESEARCH CONTRIBUTIONS...16
1.6 OVERVIEW OF THE THESIS...16
CHAPTER 2 Research Background...18
2.1 INTRODUCTION...18
2.2 XML COMMENCEMENTS AND IMPORTANCE...18
2.2.1 XML document types...*20*
2.2.2 Java API for XML (JAXP)...*22*
2.2.3 XML Retrieval...*23*
2.2.4 XML Query Languages...*25*
- XPath...25
- XQuery...27
- XLink and XPointer...27
- NEXI...28
2.3 TYPES OF QUERIES...28
2.4 VAGUE QUERIES...30
2.5 CHAPTER SUMMARY...31
CHAPTER 3 State of the Art Technology in Compressing and Querying XML Documents...33
3.1 XML COMPRESSION TECHNIQUES...33
3.1.1 Queriable XML Compressors:...*35*
3.2 PROCESSING VAGUE QUERIES TECHNIQUES...40
3.3 PROBLEM IDENTIFICATION...43
3.4 CHAPTER SUMMARY...44
CHAPTER 4 XML Compressing and Vague Querying (XCVQ) Design...45
4.1 SYSTEM ARCHITECTURE...45
4.2 XCVQ-C DESIGN...47
4.2.1 Creating the Structured-Tree & its Abridgment...*48*
4.2.2 Creating the Containers...*50*
4.2.3 Compressing the Containers...*51*
- LZW Compression Technique...52
- Gzip Compression Technique...53
4.3 XCVQ-C ALGORITHMS AND THEIR CORRECTNESS...53
4.3.1 startElement algorithm...*54*
4.3.2 endElement algorithm...*56*
4.3.3 endDocument algorithm...*57*
4.4 XCVQ-D DESIGN...58
4.5 XCVQ-D ALGORITHM AND ITS CORRECTNESS...60
4.6 XCVQ-QP DESIGN...64
4.6.1 XPath Query...*64*
- Path Matching Expansion...65
- Data Value Matching Expansion...69
- Function Set Expansion...70

4.6.2 Query Decomposer *71*
4.6.3 Query relaxation *74*
4.6.4 Ranking *78*
4.6.5 Decompression *79*
4.7 Chapter Summary 80
CHAPTER 5 *XCVQ* Testing, Evaluation and Discussion 81
5.1 Testing Strategy 81
5.1.1 Testing XCVQ's Behaviour *81*
5.1.2 Testing XCVQ's Structure & Functionality *84*
5.2 Testing Factors 85
5.3 Data Preparation 87
5.4 Testing Environment 88
5.5 XCVQ-C and XCVQ-D Testing 88
5.5.1 XCVQ-C and XCVQ-D Testing: Stage-1 .... *88*
5.5.2 XCVQ-C and XCVQ-D Testing: Stage-2 .... *91*
5.6 XCVQ-C & XCVQ-D Evaluation 92
5.7 XCVQ-QP Testing 95
5.7.1 QFT *95*
5.7.2 QPT *96*
5.8 XCVQ-QP Evaluation 98
5.9 Chapter Summary 100
CHAPTER 6 Conclusions and future work 102
6.1 Conclusion 102
6.2 Recommendations 104
6.3 Future Work 105
Publications 107
Reference List 108
Appendix-A: XPath's EBNF 117
Appendix-B: Implementing XCVQ 120
Appendix-C: XML Corpus & XPath Benchmark 134
Appendix-D: Testing Results 139
Appendix-E: XPath Query Evaluation Benchmark 141
Appendix-F: Independent testing 144
Appendix-G: XML dummy elements ratio 145

Copyright Statement

The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and she has given the University of Huddersfield the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulation of the University Library. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

The ownership of any patents, designs, trademarks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any productions of copyright works, for example graphs and tables (“Reproduction”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions

List of Tables

Table 2-1: Differences between Data-centric and Document-centric XML.................... 21
Table 2-2: SAX and DOM features 22
Table 2-3: Query types... 29
Table 3-1: The main differences between XML-conscious and XML-blind compressors 34
Table 3-2: The main limitations of some queriable XML compressors......................... 35
Table 3-3: A comparison between different compression techniques. 38
Table 3-4: Query types with the compression techniques process each....................... 43
Table 4-1: Compression granularity comparison. 52
Table 5-1: *XCVQ* Testing factors 85
Table 5-2: Average CR for all the tested XML compressors. 93
Table 5-3: XPathMark-FT query benchmark... 96

List of Figures

Figure 1-1: System Development Methodology (Morrison and George, 1995) 13
Figure 2-1: XPath Queries Examples. (a): CO query. (b) CAS query........................... 24
Figure 2-2: The role of XPath between other XML query languages. (W3Schools.com, 2006a) 25
Figure 3-1: The distribution of the compression techniques over the years 39
Figure 4-1: Preliminary Architecture of *XCVQ* 46
Figure 4-2: The complete design of XCVQ 47
Figure 4-3: An XML example 49
Figure 4-4: The structured-tree for the example in Figure 4-3 50
Figure 4-5: Creating containers process. (a) the path-Dictionary. (b) the container...... 51
Figure 4-6: (*startElement)* algorithm 54
Figure 4-7: *(endElement)* algorithm 56
Figure 4-8: *(endDocument)* algorithm 57
Figure 4-6: Architecture of *XCVQ-D*... 59
Figure 4-10: *XCVQ*-Decompression algorithm 62
Figure 4-11: The architecture of the query processor.. 66
Figure 4-12: S*tring-similarity* match algorithm 68
Figure 4-13: The design of *XCVQ*-Query Decomposer 72
Figure 5-1: *XCVQ* State Graph 82
Figure 5-2: SCR for the XML corpus.. 89
Figure 5-3: (a) Structure Compression Time for the XML corpus and (b) Structure Decompression Time for the XML corpus.. 90
Figure 5-4: CR for the XML corpus 91
Figure 5-5: (a) Compression Time and (b) the Decompression Time for the XML corpus. 92
Figure 5-6: Evaluating *XCVQ-C* CR... 93
Figure 5-7: Evaluating *XCVQ-C* CT 94
Figure 5-8: Evaluating *XCVQ-D* DT. 95
Figure 5-9: Testing *XCVQ-QP* Querying time... 97
Figure 5-10: *XCVQ* Query processing time against (a) XGrind and Xpress, (b) XQZip, and (c) XSAQCT. 99

List of Abbreviations

API	Application Programming Interface
CAS	Content and Structure
CO	Content Only
CR	Compression Ratio
CT	Compression Time
DOM	Document Object Model
DR	Data Ratio
DT	Decompression Time
DTD	Data Type Definition
EBNF	Extended Backus-Naur Form
EXI	Efficient XML Interchange
H	Data Compression Entropy
INEX	Initiative for the Evaluation of XML retrieval
JAXP	Java API for XML
LZW	Lempel-Ziv-Welch
NEXI	Narrowed Extended XPath I
QFT	Query Functional Test
QPT	Query Performance Test
RD	Regular Documents
SAX	Simple API for XML
SD	Structural Document
SDM	System Development Methodology
SGML	Standard Generalized Markup Language
TD	Textual Document
W3C	World Wide Web Consortium
XCVQ	XML Compressing and Vague Querying
XCVQ-C	XCVQ Compressor
XCVQ-D	XCVQ Decompressor
XCVQ-QP	XCVQ Query Processor
XML	Extensible Markup Language
XPath	XPath: a XML query language
XQuery	XQuery: a XML query language
XSLT	XML Style sheet Language Transformation

CHAPTER 1 Introduction

1.1 Introduction

The eXtensible Markup Language (XML) is a World Wide Web Consortium (W3C) recommendation which has widely been used in both commerce and research. In recent years, we have witnessed a dramatic increase in the volume of XML digital information that is either created directly as an XML document or converted from another type of data representation. The importance of XML is mainly due to its ability to represent different data types within one document, solving the problem of long-term accessibility, and providing a solution to the problem of interoperability (Al-Hamadani et al., 2009).

Due to the replication of the XML schema in each record, the XML document is considered to be one of the self-describing data files, which means that these kinds of files have a lot of data redundancy in relation to both its tags and attributes (Ray, 2001). For the above reason the need to compress XML documents is becoming increasingly dramatic. Furthermore, what has evolved is the urgent need to retrieve information directly from the compressed documents and then decompress only the retrieved information (Ferragina et al., 2006).

Because of the wide range of XML documents in use and the different kinds of users, being able to deal with all kinds of queries has become a key issue. Some of these queries may have imprecise constraints which cannot be processed directly due to the grammar restriction in the existing query languages. However, these types of queries, which are known as *vague queries*, appear to be common when the users of the XML documents have little knowledge about the document structure, or may lack the skills to write a precise and meaningful query. Another type of vague queries occurs when the query is presented without the presence of a Schema or the data type definition (DTD) of the document.

According to the relevant literature, there are a number of techniques that compress the XML documents and query the compressed version with no or

partial decompression. These techniques process almost all types of queries but not the vague queries; admittedly, there are a number of researchers now trying to process vague queries on the original XML document.

The research carried out in this thesis primarily concerns designing and implementing a new technique called XML Compressing and Vague Querying (*XCVQ)* which consists of two stages. In the first stage, it separates the data part of the XML document into several containers according to the path of that data within the document. Then each of the containers is compressed separately using a back-end compressor. The second stage processes the vague queries by decomposing them into multiple sub-queries, retrieves information from the compressed XML document according to each sub-query, combines the retrieved information according to the given query, and finally decompresses only the most relevant information.

To eliminate the amount of technologies associated with the XML documents and to make the process of compressing and retrieving information easier for the inexperienced users, *XCVQ* is designed to be schema independent in both phases of the compressor and the query processor.

1.2 Research Hypothesis and Research Methodology

This thesis is based on the following hypotheses:

1. The existing XML compression techniques can be improved to construct a new schema independent XML compressor with a higher compression ratio.
2. The redundancy in the XML documents significantly affects the size of those documents and can be reduced to more than half of the original file size.
3. The compressed XML document can be retrieved according to vague queries. Vague queries are those queries which do not follow the semantic rules of current query languages. They occur when the exact matching user's query does not retrieve the required information either because of the lack of experience in writing a query or the absence of the document's schema.

4. The necessity of retrieving information from more than one XML documents without the need to specify an exact relative document.

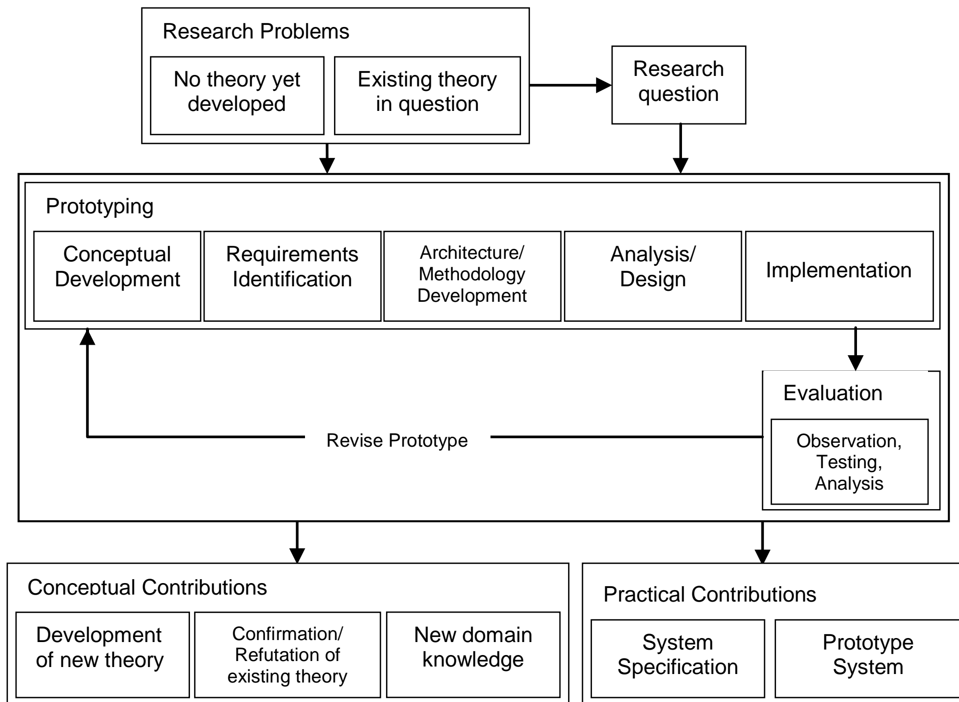


Figure 1-1: System Development Methodology (Morrison and George, 1995)

The above hypotheses are tested throughout this research by using the System Development Methodology (SDM) (Nunamaker et al., 1991; Morrison and George, 1995; Hevner et al., 2004). This methodology has been widely used by software developers and information system specialists (Meersman et al., 2008; .Yousof et al., 2011) As depicted in Figure 1-1, this methodology consists of four main stages:

1. *Identifying research problems:* This stage focuses on drawing up the research questions due in part to the lack of theories in the research field and/or build upon existing theories. In this thesis, the research questions are set from two XML fields, compressing the XML documents and querying them. As a result, a new XML compressor is introduced (CHAPTER-4) with the ability to retrieve information from the compressed document according to vague queries. The designed system

may improve the querying process to retrieve information from XML documents.

2. *Prototyping and evaluation:* In the second stage, SDM spotlights the implementing or prototyping the proposed system. It starts by designing the conceptual model of the proposed system, identifying the necessary requirements, designing the complete architecture of the system, and then implementing the system to prepare it for the evaluation process by testing and analysing it. In this thesis, the complete architecture and the detailed design of the system are laid out in CHAPTER-4, the implementation part is in Appendix-F, and the evaluation process is given in CHAPTER-5.

3. *Conceptual and practical contributions:* As a final stage, SDM sets the main contribution to the knowledge. In this thesis all the contributions, conclusions, and future developments are presented in CHAPTER-6.

1.3 Research Questions

Following the SDM as shown in Figure 1-1, outlining the research questions should be made before proceeding further with defining the actual prototype. The research into this thesis focused on two main parts, each of which has its own set of questions to be addressed:

1. Is it possible to design a new compression technique that has the ability to compress XML documents and achieve a better compression ratio without the need for the document's schema or its DTD?
2. What is the influence of the structure redundancy on the overall size of the XML document?
3. What are the main types of vague queries and when could they occur? Has the existing XPath query language the ability to answer vague queries? If not, what is the required expansion that should be made on XPath to provide it with such ability?

4. How does one determine the relevant XML document(s) from thousands of documents without the need to scan them completely for time saving purposes? And is it possible to retrieve information from more than one XML document without the pre-specification of these documents using one XPath query?

1.4 Motivations and Objectives

This work is initially motivated by the need to expand the XML query languages. These languages are treating the user's query in Boolean nature (Campi et al., 2009) in which a specific XML node is selected if and only if it satisfies exactly the query or part of it. This case applies more restrictions to the inexperienced user or in the case of schema absence.

XML has become a focus for research in both the database as well as the document research communities (Harrusi et al., 2006; Moro et al., 2008). This research is motivated by the strength of XML such as its simplicity, the separation of data from the structure, interoperability, and human and machine readability. All these features and more make the XML document a reliable way for data transformation on the web. However, the redundancy in the structure of the XML documents enlarges their sizes, the very reason that inspired researchers to produce compression techniques dedicated to XML. Other researchers were interested in retrieving information from the compressed XML document to make it easier to use these fairly large documents with low resource devices. Although these techniques succeeded in answering several types of queries, they are incapable of processing vague queries, which is yet another motivation for this research.

The main objective of this research is to investigate the different types of vague queries and set new methods to solve these queries in the case of existing compressed XML document. The design and implementation of a system that has the ability to compress the XML document and retrieve information from the compressed file according to vague queries, let alone the need to decompress only the retrieved relevant information, is another objective of this research.

Since it was very difficult to have access to the source code of an XML compressor to be used as a first stage to achieve the main objective, another objective therefore was to design and implement a new XML compressor that has the ability to achieve a better compression ratio than the existing techniques.

1.5 Research Contributions

This research will contribute to the fields of XML compression and XML retrieving in the following areas:

1. A new XML compression technique is introduced that compresses XML documents efficiently and independently from their Schema or the DTD. The designed compressor achieved a compression ratio of 1.83 which is higher than the best existing techniques.
2. Identify the exact ratio of the redundancy of the XML structure. This redundancy is abridged by up to half the size of the original file.
3. The main contribution of this research is the introduction of a new method to answer vague queries, a kind of queries that can be submitted by naive users or via the absence of the document's schema. The new method is adjusted to process the vague queries under the compressed XML documents and retrieve the most related results.
4. Introduce the idea of retrieving information from XML documents without specifying the exact documents that have the required information.

1.6 Overview of the Thesis

Apart from Chapter 1, the thesis has six more chapters:

Chapter 2: Research Background. This second chapter sets forth the research background including all the techniques used in the research process. The most important features of XML documents are listed, accompanied by their types, the API used to parse them, the techniques used to retrieve information from

them, and their query languages. This chapter also lists all the query types and provides a complete definition of vague queries.

Chapter 3: State-of-the-Art Technologies. This chapter is separated into two main parts. The first one concentrates on discussing the main XML compression techniques and sets the differences between them, their advantages and their drawbacks. The second part discusses the techniques that have been used to solve vague queries from XML documents.

Chapter 4: *XCVQ* Design. This chapter illustrates the design of the *XCVQ* system starting with the main architecture of the complete system. Then it sets the detailed design of the compressor, followed by the design of the decompressor. This chapter ends by giving the complete design of the vague query processor. It is supported by the algorithms that are used to answer the research questions.

Chapter 5: *XCVQ* Testing, Evaluation and Discussion. This chapter sets all the testing process for the *XCVQ*-compressor to obtain the compression ratio and for the *XCVQ*-query processor to determine its functionality and the performance. An extensive test has been done to compare *XCVQ* with the other existing techniques. All the results of these tests can be seen in this chapter. It ends with the discussion part that illustrates the main features, advantages, and drawbacks for the designed system.

Chapter 6: Conclusion and Future Works. This chapter summarizes the main conclusions and contribution of the research and suggests more development and expansion for further research.

CHAPTER 2 Research Background

2.1 Introduction

This chapter provides the background to our research. It comprises several key parts. The first one illustrates the most important techniques that motivate and support this research. XML, being the most important key technology, is presented in this section, alongside the structure of the documents which would be crucial in the design of the compressor. XML query languages and the main differences between them are also discussed because of their importance in retrieving information from such documents.

2.2 XML Commencements and importance

Before the rise of the internet, 1980s witnessed the invention of Standard Generalized Markup Language (SGML) as a way to display information dynamically. Later, in 1995, W3C recommended SGML to be used for the internet. Problems occurred when using SGML included the lack of widely supported style sheets, complexity and instability in the software that were using it, and the difficulties in interchanging SGML data due to its varying levels among SGML software packages.

In 1996, the first XML working draft was intended to be a powerful substitute to SGML. It was first recommended by the World Wide Web Consortium (W3C) in 1998 to be used as a mark-up language for storing and exchanging data through the web. The most recent recommendation was published in 2008, which is the fifth edition of the XML (W3C, 2008). In a very short period of time, XML has become the basis for data exchange through the Internet. This is due to its several features such as the following (NG et al., 2006; Gerlicher, 2007; Groppe, 2008):

- Readability: XML is readable by both human and machine. This means that the data represented by XML can be used by different users and by different parsing code.

- Interoperability: This is the ability of the hardware and software to use XML documents without the need to make any changes to the software or the data itself. This means that XML data is stripped of any dependency on software and machine.

- Long term usability: Since XML documents are represented using the Unicode; these documents are expected to stay in secure storage and usage for years (Augeri et al., 2007; De Meo et al., 2007) .

- Extensibility: This means that there are no fixed set of tags that should be used to represent data.

- Generality: XML documents have the ability to represent different kinds of data representation such as images, sounds, videos, texts, etc.

- Internationality: Almost all written languages can be represented in XML documents since they support Unicode (Norbert and Kai, 2004).

In spite of all these advantages, XML has also some weaknesses:

- They have a huge amount of redundancy which makes these documents demand high storage memory to be archives, high band width to be transmitted, and high cost to be processed.
- The huge amount of technologies surrounding it complicates the use of these documents such as schema, DTD, XSLT, SAX, DOM, XPath,

XQuery. These technologies render the use of these documents somewhat difficult especially with naive users or in cases where these technologies are absent, it would be just as difficult as they are considered necessary for dealing with XML documents.

- The problems that can occur when dealing with the document namespace should be carefully sorted out otherwise other problems and complications could occur during the processing of these XML documents.

2.2.1 XML document types

The main building blocks of any well-formed XML document are nested open tags and their equivalent close tags. These tags can be formed as follows (Hunter, 2000; Anders, 2009; Goldberg, 2009):

1. *Elements:* each element starts with an open tag (<p>) and ends with an end tag (</p>). Everything between and including these tags are an element. The general structure of an element is as follows:

   $$<e\ \ at_1=\text{"}v_1\text{"}\ \ at_2=\text{"}v_2\text{"}\ \ at_n=\text{"}v_n\text{"}>d_1d_2d_3\ldots d_m</e> \quad (1)$$

   Such that $n \geq 0$, *and* $m \geq 0$

   Each element has an *element-name* (e) which should follow the following rules:

   - Case sensitive names.
   - Consist of characters, numerals, underscores and tabs.
   - Start with a character or an underscore.
   - Should not start with *xml* or *XML*.

   Elements can have optional *element-value* ($\{d_1d_2d_3\ldots d_m\}$ in (1)) which represent the actual data values for the XML document.

2. *Attributes:* attributes (if any) appear within an element and they provide more information about that element. Each attribute has an *attribute-name* ($\{at_1, at_2, at_n\}$ in (1)) which should follow the same rules for an

element-name, and an *attribute-value* ($\{v_1, v_2, \ldots v_n\}$ in (1)) which can be any printable character between a pair of quotations.

3. *Data text:* the data in a XML document could either be *attribute-values* or *element-values*. This text can be a list of any keyboard printable character from the Unicode set ($\{d_1d_2d_3 \ldots d_m\}$ in (1)). Some escape character should be used to embed some of the characters in the data text such as (*<*), (*>*), (*&*), (*"*), and (*')* to represent (<), (>), (&), (“), and (‘) respectively.

4. *Comments:* comments can be added anywhere in the XML document to provide any further description but is not part of the main document. In XML, the comment start tag is (<!--) and the end tag is (-->).

5. *Declaration:* this single statement (if any) should be the very first line of the document. It supplies the XML processor with information such as the version, encoding and other information about the document. Its start tag is (<?xml) and its end tag is (?>).

Table 2-1: Differences between Data-centric and Document-centric XML

Criteria	Data-centric	Document-centric
XML role	Superfluous	Significant
Order	Not very important	Significant
Consumption	Machine	Human
Data granularity	Fine	Large
Examples	Catalog and flight schedules	Books and advertisements

Depending on the amount of data (*attribute-values* and *element-values*) in the XML document, Bourret (2005); and Manning et al. (2008) classified XML documents into two types, either data-centric or document-centric. Table 2-1 lists the main differences between these two types according to certain criteria.

With data-centric XML documents the roles of the XML elements and attributes are to arrange these data in atomics. These documents are usually created and used by machines such as the XML documents that are generated by a Database Management System, or those used to transfer data between different databases.

In contrast, XML role in document-centric XML documents is very important since it is the only way to organize this document into large units of information. The order of the elements inside these documents is important since any change in the order can produce a completely different document.

2.2.2 Java API for XML (JAXP)

Java programming language, and some other languages, provides different types of XML Application Programming Interface (API) such as SAX, DOM, and XSLT (Violleau, 2001; McLaughlin and Edelson, 2006; Williams, 2009) in order to process the XML documents by means of writing a computer programme using several programming languages. SAX (*Simple API for XML*) scans the XML document sequentially and throws up events that the programmer can handle. These events are thrown by the parser when it detects the start-document, end-document, start-element including a list of all its attributes, end-element, and characters. The programmer should write suitable codes for each event to process an entire XML document. Since each event occurs only once for each element, all the required work needed to process the document should be done in one cycle.

Table 2-2: SAX and DOM features

SAX	DOM
Event based model	**Tree-like structure**
Sequential access	**Random access**
Required low memory	**Memory intensive**
One scan for the document	**Multiple traverse for the document**
1998, David Megginson's	**1998, W3C's**

By using DOM (*Document Object Model*) parser, the document is represented in the main memory of the computer as a tree-like structure. The programmer can write the code to traverse this tree as many times as s/he needs. Table 2-2 sets out the main features of SAX and DOM. It shows that using DOM parser is memory consuming and since the aim of this research is to reduce the amount of memory used to process the XML documents, the designed system used SAX parser to process these documents.

While SAX and DOM parsers should be used through a programming language, XSLT (*XML Style-Sheet Language Transformation*) is a declarative language which is used to transform the XML document into another document type (Tidwell, 2008; Williams, 2009). Its two main purposes are: (1) produce HTML documents from XML documents for browsing purposes, and (2) retrieve information from the XML document using the XPath.

2.2.3 XML Retrieval

XML retrieval is considered to be one of the semi-structured retrieval techniques (Manning et al., 2008). This adds more challenges to meeting the user's needs. The first difficulty in structured retrieval is that the user requires only parts of the documents and not the entire document like unstructured retrieval techniques do (Stamatina et al., 2006). This challenge leads to another, which is the identification of the most relevant parts from the document to the user's query. To solve this difficulty there are two approaches, either to retrieve the largest units of the document that contains the required information (top down) (Norbert and Kai, 2004; Jiaheng, 2006), or to retrieve the smallest unit by starting the search from the leaves of the XML tree (bottom up) (Fuhr et al., 2006).

Retrieving information from XML documents provides the users with the extra abilities to specify the exact piece of information needed or to combine different parts from of the document that meet the user's need. The user's queries can specify the required information as well as the place where this information is to be found inside the document. For instance the user may ask

about *"a table of all XPath functions in XPath description chapter"*. In this case the *"XPath functions"* and the *"XPath description"* are about the content of the document, while the *"table"* and the *"chapter"* are about its structure.

XML documents can be retrieved according to their type either text-centric or data- centric retrieval. In text-centric, an approximate matching process is used to match the text of the query with the text of the document while the structure role is as a framework within this process (Manning et al., 2008). Since the matching process is done with the data part of the XML document, the retrieved information is expected to be long and they should be ranked. On the other hand, data-centric retrieval retrieves only attribute values and numeric data using exact match. The retrieved information from this type is short and the ranking is not significant.

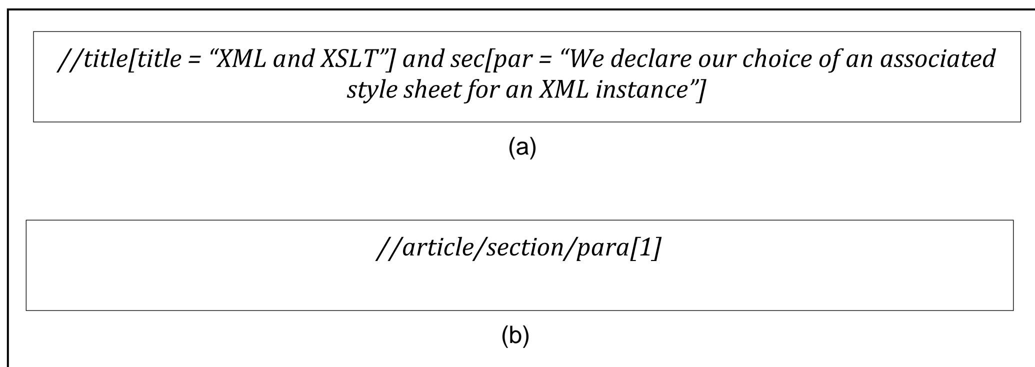


Figure 2-1: XPath Queries Examples. (a): CO query. (b) CAS query

Another classification for XML retrieving techniques is done according to which part is more significant in the user's query: the content part or the data part (Hunter, 2000; Sanz, 2007). Content-Only (CO) queries are rich of text and focus on the data part of the XML document. The user can add some structural constraints to the query to specify the granularity of the required information. As seen in Figure 2-1-(a), the XPath query focuses on retrieving the title and the content of a paragraph which is considered the data content of the document. To process these queries, some of the techniques use the traditional IR techniques by completely ignoring the structural constraints and treat the XML document as

a traditional text file, while other techniques decompose the query into several small queries and process each one separately.

Content-And-Structure (CAS) retrieval takes into their considerations the structural part of the XML document and provides the user with extra advantages to accurately specify the exact part required from the relevant document. The XPath example in Figure 2-1-(b) concentrates on finding the first paragraph which is in a section for the specific article.

2.2.4 XML Query Languages

Different kinds of query languages have been proposed in order to retrieve specific information from an XML document. All these languages have a common feature in that the user should specify the exact XML document(s) wherefrom s/he would like to retrieve the information. This section spotlights the main features of some of the query languages which are either recommended by W3C (XPath, XQuery, XPoint, and XLink) or used by Initiative Evaluation of XML retrieval (INEX) working group (NEXI).

- XPath

Standing for XML Path language, it is a descriptive language which takes an XML document and a user query as an inputs and produces specific nodes from this document as output (Kay, 2004).

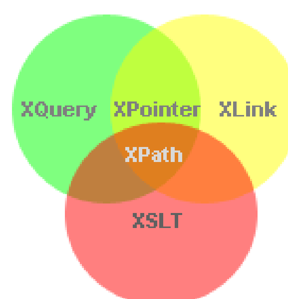


Figure 2-2: The role of XPath between other XML query languages (W3Schools.com, 2006a)

It is considered to be the core to all other XML query languages, as illustrated in Figure 2-2 (W3Schools.com, 2006a). The main building blocks for an XPath expression are: (1) expressions deal with atomic values which include comparative and arithmetical operations, (2) expressions for selecting specific nodes from a tree, and (3) operation on every item in a specific sequences, such as using the *"for"* expression.

In 1990, W3C recommends XPath 1.0 as an XML query language (W3C, 1999). In 2010 W3C recommend the last version of XPath 2.0 to be a standalone query language or to be embedded with XSLT or XQuery (W3C, 2010a). It comes with some developments on the first version. These changes make XPath easy to use, improve its interoperability, simplify the manipulation of string contents and Schema-typed content, and to increase its efficiency. These developments include: (Holman, 2002; Kay, 2004; W3C, 2007b; Kay, 2008)

1. Data types: XPath 2.0 offers new data types such as integers, single precision, date, time, and any data type that can be defined by the user through XML Schema.
2. Path expressions: Not very big changes on path expression compared to XPath 1.0, only the ability to use the function call within the path expression is a slight change.
3. Operators: addition operators are used to support XPath 2.0 functions. Examples of these operators are: *"is"* to test if two expressions return the same set of nodes, *"<<" and ">>"* to test the order of the two operands, *"except" and "intersect"* to find the difference and the intersection between two node sets, and *"eq", "ne", "lt", "le", "gt", "ge"* to make a comparison between atomic values and return a node set.
4. Functions: Some new functions are added to the list of the available ones in XPath 1.0 such as: *"max()", "min()", "avg()",* functions to manipulate the new data types like *date*, *time*, and *QNames*, generalization of string manipulation functions to deal with user-defined types.

   Path expressions thus provide a very powerful mechanism for selecting nodes within an XML document, and this power lies at the heart of the XPath language (Sigurbjornsson and Trotman, 2003; Kay, 2004).

- XQuery

In 2007, W3C first recommended XQuery as an XML query language and they made the last recommendation in 2010 (W3C, 2010b). This querying and descriptive language uses XPath to retrieve information from XML documents whereas the simplest XQuery expression is an XPath expression. The main engine in XQuery is the *"FLWOR"* expressions which stand for For-Let-Where-Order-Return. In these expressions, the *"for"* expression selects a specific node list from a specific document which can be repeated several times within the same expression. The *"let"* expression associates with each node in the node list(s) generated by the first expression or another node retrieved from another XML document. The *"where"* expression filters the resulting list according to a specific condition. The *"order"* expression sorts the list according to a specific atomic. Finally, the *"return"* expression specifies the required information from the node list(s).

The main advantage of XQuery over XPath is that XPath by its own cannot organize the output of the query in a specific format while XQuery does (McGovern et al., 2003). Although XML Style sheet Language Transformation (XSLT) can do organize the format of the retrieved information, but it could be difficult for the user to use it due to its recursive-structure and mixed name-spaces.

Another feature in XQuery is its capability to retrieve information from more than one specified XML documents. XPath is suffering from the lack of this feature.

Although it is considered to be very easy to use, XQuery is a read-only query language. This means that XQuery does not have the ability to exchange or create an XML document like SQL to the databases.

- XLink and XPointer

XLink stands for Linking Language and recommended by W3C in 2001 (W3C, 2001). The main purpose of XLink is to make either *"simple"* links

between two XML resources, or *"extended"* links between more than two XML resources (W3Schools.com, 2006a). With the *"simple"* links, any element inside the XML document can be linked with another resource such as an image, a text file or even another XML document just like the *"a"* element in HTML which performs Unidirectional link. When the link type is *"extended"* this means that the link will be bidirectional between the XML document and the other resource(s).

XPointer stands for XML Pointer Language and recommended by W3C in 2003 (W3C, 2002). It uses XLink to point to specific data part within the XML document. This means that XPointer query should starts with the URI of the document followed by *"#"* sign which indicates the starting of the XPointer query which is actually an XPath query with some extra functions.

- NEXI

Stands for Narrowed Extended XPath I is an XML query language that follows the steps of XPath with some modifications. First, the NEXI retrieval engine designed to deduce the semantics from the query in reverse to XPath which has predefined semantics. Furthermore, NEXI extended the use of the contains() function, which is used by XPath to indicate an element that is contain a specific content, to be about() function to indicate the element to be about the content. This adjustment allows NEXI to deal with fuzzy queries. NEXI has been used for several purposes, such as question answering, multimedia searching, and searching heterogeneous document collections. (Trotman and Sigurbjornsson, 2005)

2.3 Types of Queries

Queries are questions written by users to search, change or retrieve a specific piece of information from different types of files such as text, image, or database files. Depending on the query functionality, they can be categorized into three types. The first type is the *selection queries* which are responsible for selecting and retrieving the relevant document or sub-documents and returning the results to the user. The *action queries* are the second type. These queries implement a

specific action on the selected file or document, such as delete, add, or update a piece of information. The third type is the *aggregate queries* which find the statistical amount for the selected attributes such as average, max, min ...etc.

Table 2-3: Query types

Query type	**Description**	**XPath Example**
Simple queries (SQ)	Retrieve part of the document according to general specification	List countries names *//countries/country/name*
Criteria queries (CQ)	Retrieve part of the document according to a specific criterion.	List countries with less than 10 million population *//countries/country[population < 10000000]*
Conjunctive queries (JQ)	Retrieve part of the document according to conjunction of two or more criteria.	List industrial countries with less than 10 million population *//countries/country[economy=”industry”] and //countr[population < 10000000]*
Range queries (RQ)	Retrieve information according to a range between given minimum and maximum values.	List countries with population between 6 million and 15 million *//countries/country[population > 6000000] and //countries/country[population < 15000000]*
Vague queries (VQ)	Retrieve information when there is no Boolean matching between the user’s query and the relevant XML document (Stasiu et al., 2005; Rajpal et al., 2007).	List countries with population between 6 million and 15 million */country/population between(6000000, 15000000)*

Depending on their complexity, selection queries can be categorized into five main types. Table 2-3 lists these types and describe their features supported by an example for each type and its equivalent XPath query (written in *Italic* in the table). The amount of the retrieved information varies according to the query’s

level of complexity such that the simplest query retrieves general information while the more complex query tries to retrieve more specific information.

Since vague queries are the central issue in this research, the following section provides a brief description of such queries and how they can appear in information retrieval domain.

2.4 Vague Queries

XML query languages force the users to follow their rigid rules to write a syntactically true query. This process is not easy to be maintained even for expert users (Huh et al., 2000). Moreover, in order to retrieve the required information these languages require previous full knowledge about the document's schema what is considered to be difficult to ordinary users. If the query does not follow the semantic rules of the querying language or it does not meet the document's schema, null information will be retrieved because these query languages use Boolean conditions wherein a condition is either true (exact match) or false (no match) (Campi et al., 2009). On the other hand, handling the fault-tolerant for the user's query makes it easier for the user to retrieve approximate information when vague conditions appear in the query (Zhao and Ma, 2009).

Vague queries are those that occur when exact matching queries fail to retrieve the required information (Fuhr, 1999; Bodenhofer and Küng, 2001; Zhang and Kankanhalli, 2003; Dutta et al., 2009). In this case the vague query needs to be generalized to retrieve the relevant information and rank this information in the bases of their relevancy. Vague queries can be cause by several factors:

1. *Schema*: although XML Schema or its DTD are very important when creating and developing the document, their absence during the retrieving process leads to null information retrieved since all XML query languages demand complete knowledge over them. Even if the schema exists, it is difficult to figure out the exact structure of its XML document (Sakr, 2009; Al-Hamadani et al., 2011).

2. *Users*: there are two main kinds of XML retrieval users, the experts and the naïve. The experts have the ability to write syntactically true queries depending on their knowledge of the rules of the query language and have the ability to navigate the document's schema and write the appropriate query. However, different kinds of XML schema available such as XML Schema, DTD, and RNG, and even expert users are only aware of one or two of them. On the other hand, the naïve users have low experience in the rules of the language and in the schema navigation. The latter case could produce vague queries which has spelling errors in either the structure or the content of the document, different case used in the query and in the original document, or out of order or weak path (Florescu et al., 2000; Campi et al., 2009).

3. *The query Language*: all XML query languages do not have the ability to retrieve approximate answers according to a user's query. Moreover, the functions in the query languages sometimes do not meet the user requirement. All these restrictions in the languages can lead to vague queries (Buneman et al., 2003; Norbert and Kai, 2004).

4. *Unknown document*: whenever a query is submitted, it should specify the XML document(s) that has the required information. If the user does not know the exact document or the information is disseminated in more than one document, a vague query occurs (FAZZINGA et al., 2009).

2.5 Chapter Summary

This chapter described the origins of the XML technique and its development. It showed the importance of the XML documents and their usage as well as their drawbacks. Since these documents have a special structure, this chapter provided a brief description of this structure and the different types of documents. To deal with XML documents, many APIs have appeared. This

chapter listed the well known APIs and described their features and differences. Because this research lies in the field of XML retrieval, the chapter highlighted different kinds of XML retrieval techniques and query languages used to retrieve parts of the entire XML document. The main features of all types of queries are illustrated with the focus being on vague queries.

CHAPTER 3 State of the Art Technology in Compressing and Querying XML Documents

Since this research consists of two main parts, the XML compressor and the vague query processor, this chapter discusses the main XML compression techniques in its first part. It will highlight the advantages and disadvantages of these techniques and discusses the differences between them. The second part of this chapter will focus on the vague query processors used to retrieve information from XML documents.

3.1 XML compression techniques

Recently, large numbers of XML compression techniques have been proposed. Each of which has different characteristics. This section discusses the differences between these compressors and their main features.

XML compressors can be classified into two classes either to be *XML-blind* or *XML-conscious* compressors. XML-blind or general purpose compressors deal with the XML document as a traditional text document ignoring its structure and apply the general purpose text compression techniques to compress them. These techniques can be classified into two main classes (Salomon, 2007), either to be statistical or dictionary based compressors (Augeri et al., 2007; Augeri, 2008). The statistical or the arithmetic compressors represent each string of characters using a fixed number of bits per character. PPM, CACM3, and PAQ are examples of this kind of compressors (Cleary and Witten, 1984; Moffat., 1990; Alistair et al., 1998). On the other hand, dictionary compression techniques substitute each string in the input by its reference in a dictionary maintained by the encoder. WinZip, GZIP, and BZIP2 are examples of this compression class (WinZip, 1990; GZip, 1992; BZip2, 1996).

Table 3-1: The main differences between XML-conscious and XML-blind compressors

XML-conscious compressors	XML-blind compressors
Information about XML documents is usually available in schema which can be optimized by XML-conscious compressors to get better compression.	Cannot take advantage of the schema to get useful information about the file.
They utilize the structure of XML document and the type of the data inside.	They do not take in consideration the entire file structure or data types.
Some of them abridge the original XML tree in a summary or compact tree for better ratio.	They cannot exploit redundancies in the XML tree structure.
Most of them are powerful in compressing small or large files.	They do not efficiently compress small files that can be used in transactions for e-business. (Hung, 2009)

On the other hand, XML-conscious compressors try to utilize the structural behaviour of XML documents in order to achieve better compression ratio and less time in comparative with the XML-blind type. Table 3-1 sets the main differences between the two aforementioned compressors types.

The main theory of data compression, which described in (Shannon, 1948), is the formulation of the entropy rate *(H)* which indicates the limit to lossless data compression. The value of *(H)* depends on the probability of each symbol in the information source. The most popular entropy value is:

$$\boldsymbol{H = \sum_{i=1}^{n} P_i log \frac{1}{P_i}} \qquad \text{(Shannon, 1948)} \qquad (2)$$

Where, $\boldsymbol{P_i}$ is the probability of the symbol $\boldsymbol{a_i}$.

In this paper, Shannon proved that the compression ration cannot exceed the value of *(aH)*, where *(a)* is the number of symbols in the source.

Since XML are heterogeneous data, the theory of XML compressors is to separate the data from the structure, separate the data into containers according to the type of the data, and apply a general purpose compressor for each container. This process can lead to produce an optimal compressor over heterogeneous data. (Liefke and Suciu, 2000) developed the entropy value for XML compression to be:

$$H = \frac{1}{2}(H_0 + p_1 H_1 + \cdots + p_k H_k) \quad \text{(Liefke and Suciu, 2000)} \qquad (3)$$

Where, $H_0, H_1, \dots, H_k$ are the entropies for the sources, and $p_1, p_2, \dots, p_k$ are the probabilities of these sources.

XML-conscious compressors can be classified according to their ability to querying the compressed documents into two main sub-classes; these are queriable and non-queriable compressors. While the queriable compressors have the ability to retrieve information from the compressed XML document without the need to completely decompress the document, the non-queriable XML compressors are used to compress the XML documents for archival purposes only and they achieved better compression ratio than the queriable compressors.

3.1.1 Queriable XML Compressors:

The main goal of this type of compressors is to provide the ability to the compressed version of the XML document to be queried without complete decompression them. The compression ratio for these compression techniques is lower than the blind-XML or the non-queriable techniques.

Table 3-2: The main limitations of some queriable XML compressors.

Compression technique	Limitations
XGrind	**o Requires partial decompression to handle range and partial-match queries.** **o Lower compression ratio comparative with other compressors.**
XPress	**o Limited experimented data corpus to depth 5 and 6 only and large documents (>12MB).** **o Handles only exact-match, partial-match, and range queries.**
XQzip	**o Ignoring IPs and comments from being compressed.** **o Critical in choosing the appropriate block size to balance between the good compression ratio and efficient query processing.** **o The need for partial decompression to handle string matching queries.**
XQueC	**o Using too many structures with their pointers which yield to huge space overhead.** **o Long compression and decompression time.**
XSAQCT	**o Lossless compressor since it does not taking into consideration the order of the attributes in an element.** **o Queries only the exact match queries.**

SXSI	o Designed to increase the querying speed. o The compression ratio has not been tested. o Supports only navigational queries and string matching predicates.

However, these techniques are important when dealing with resource-limited applications and mobiles. Some of these techniques are homomorphic compressors, which mean that the compressed file is a semi-structured one. In the next section, a brief description of some of these techniques will be given, and Table 3-2 explains their main limitations.

The first queriable compressor is *XGrind* by (Tolani and Haritsa, 2000). This technique replaces the elements and attribute names with the letters *"T"* and *"A"* respectively, followed by a unique identifier which represents the substituted element or attribute name. Moreover, it replaces the end tags with *"/"* sign. The data part of the document is encoded using Huffman encoding. For the purpose of querying the compressed document, XGrind's query processor finds the simple path to check whether it satisfies the path in the given query. The main drawback with XGrind is that while it has the ability to process exact-match and prefix-match queries on the compressed documents, a whole range of or partial-match queries require partial decompression to be handled.

In order to solve *XGrind's* partial decompression problem, *Xpress* (Min et al., 2003) uses the *reverse arithmetic encoding* method to encode the label paths of the XML document as a distinct interval in [0.0, 1.0) . Using the relationships between these intervals will allow for the ability to evaluate path expressions more efficiently on the compressed XML document. Furthermore, by using this method, *XPress* uses path-by-path matching instead of element-by-element matching that has been used in *XGrind*. To encode the data part of the XML document, *XPress* uses different compression techniques depending on the type of the data and without the need to the human interference. (Min et al., 2009)

Because *XGrind* and *Xpress* are homomorphic, the relationship between the size of the compressed document and the size of the original one is linear. To solve this problem (Cheng and NG, 2004) proposed a new technique (*XQzip*) that depends on extracting the *Structure Index Tree (SIT)* from the tree structure of the original document. The SIT depth is non-linear to the structure tree which

makes this technique accomplishes higher compression ratio and faster query evaluation.

Instead of using (SIT), *XQueC* (Arion et al., 2007) uses the *structure summary tree* in order to efficiently stores the XML documents. The space needed to store the structure summary (*SS*) is:

$$\boldsymbol{CSaux} = \sum_{n \in SS}(|\boldsymbol{tag}(\boldsymbol{n})| + \mathbf{log_2}(|\boldsymbol{SS}|)) \quad \text{(Arion et al., 2007)} \qquad (4)$$

This represents the summation of the space needed to store a tag node plus the space needed to store all its successive nodes. Furthermore, instead of using hash table to store the tags and attribute names, *XQueC* used the *structural identifiers*, which has been used in some querying techniques (Al-Khalif A et al., 2002; Grust, 2002; Halverson et al., 2003; Paparizos et al., 2003) in order to uniquely identify a node in the XML tree. This technique considered to be the first one that uses XQuery as a query language.

In their work, (Müldner et al., 2009) created an annotation tree to succinctly store the structure of the XML document and use the containers to store the data part of the document. Their compressor, named *XSAQCT*, has two versions; the first was dependent on the XML Schema and the second was schema-free. They showed that the first version is better than the second from the standpoint of compression ratio even though it was slower.

Finally, (Arroyuelo et al., 2010) proved in their proposed *SXSI* compressor that the XPath queries can be performed better when using an indexing technique to compress the XML document. This technique is based on producing a labelled tree from the XML Tree structure and then indexing this tree into a bit array and compressing the data part of the document using a general back-end compressor. Although the compression ratio of SXSI is not calculated, the querying time and the retrieving quality are better than traditional retrieving techniques.

Table 3-3: A comparison of different compression techniques.

Compression technique	XML-Conscious	Schema dependant	queriable	Compression technique	Back-end compressor	Average compression ratio
WinZip (WinZip, 1990)	No	No	No	Reducing algorithm+ AES encryption	-	0.48
BZip2 (BZip2, 1996)	No	No	No	Burrows-Wheeler+ Huffman	-	0.24
GZip (GZip, 1992)	No	No	No	LZ77+ Huffman	-	0.36
XMill (Liefke and Suciu, 2000)	Yes	No	No	Dictionary-based	Gzip, Bzip2, PPM	0.55
Millau (Girardot and Sundaresan, 2000)	Yes	Yes	No	Dictionary-based	GZip, deflate	0.58
xmlppm (Cheney, 2001)	Yes	No	No	Statistical models	PPM	0.57
dtdppm (Cheney, 2005)	Yes	Yes	No	Statistical models	PPM	0.58
XWRT (Skibinski et al., 2007)	Yes	No	No	Dictionary-based	Gzip, LZMA, PPM	0.54
RNGzip (League and Eng, 2007)	Yes	Yes	No	Deterministic automaton Tree	Gzip	0.58
LXC (Bonifati et al., 2009)	Yes	No	No	words abbreviation	-	0.59
XGrind (Tolani and Haritsa, 2000)	Yes	No	Yes	Dictionary-based	Huffman	0.57
Xpress (Min et al., 2003)	Yes	No	Yes	Dictionary-based	Reverse encoding	0.57
XQZip (Cheng and NG, 2004)	Yes	No	Yes	Dictionary-based	Gzip	0.66
XQueC (Arion et al., 2007)	Yes	No	Yes	Binary encoding	Depending on the type of data	0.68
XSAQCT	Yes	Yes	Yes	Tree-size	Bzip2, gzip,	0.80

(Müldner et al., 2009)				elimination	PAQ8	
SXSI (Arroyuelo et al., 2010)	Yes	No	Yes	FM indexing	BWT	n/t

Table 3-3 shows the main differences between the various compression techniques mentioned above. It is clear that the compression ratio of all XML-conscious compressors are better than traditional blind-compressors and the compression ratios of the queriable compressors are still less than those of non-queriable techniques.

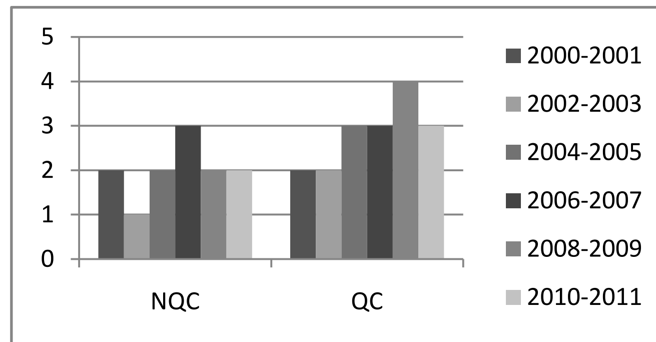


Figure 3-1: The distribution of the compression techniques over the years

Figure 3-1 demonstrates the distribution of the compression techniques over the years, where NQC and QC refer to the non-queriable and queriable compressors respectively. It shows that the years 2006 and 2007 witnessed an increasing amount of compression for both queriable and non-queriable techniques. The overwhelming rise in the number of queriable-XML compressions in the years 2008 and 2009 reflects the importance of this type of compressors.

3.2 Processing Vague Queries techniques

In order to elevate the flexibility of querying XML documents, many researchers have produced varied approaches to meet that need.

In their work (Damiani and Tanca, 2000) proposed a technique to solve what they called "blind queries" which refers to the queries submitted with the absence of XML schema. This technique first transforms the XML document into a labelled graph and provides each node a number which represent the importance of this node in the XML document. The graph then is expanded to perform a fuzzy graph. To process the vague queries, it creates a graph for the query and performed a similarity match between the two graphs.

According to the importance of approximate retrieval, (Schlieder, 2001) proposed a query language named *approXQL* since the existing XML query languages have the ability to answer queries according to exact matching only. This language is designed to answer vague queries on data-centric XML documents by encoding them into a labelled tree. It uses three pointers to encode each node of the document's tree by associating it with its pre-order number, the number of its ancestors, and the pre-order number of its most right leaf. To answer vague queries, it makes useful node transformation on them which are insertion, deletion and renaming. Each transformation is associated with its cost and the results that require less transformation cost are the most relevant once.

Instead of expanding a query language, (Amir-Yahya et al., 2002) proposed a new algorithm that depends on converting the XML document and the user's query into a tree-like structure and perform some relaxation process on the latest tree by deleting, inserting and renaming the nodes in that tree to be matched with the original XML document. Each of these processes attached with a score in order to compute the Top-k relevant answers. This technique solves the problem of generating large amount of sub-queries when using the query re-writing algorithm which has been used in *approXQL*. In this paper, the authors debate applying the traditional IR techniques to retrieve approximate answers and they prove that those techniques are not sufficient enough when dealing with XML documents, however, converting the document to a tree-like structure and applying approximate matching on it is more appropriate.

In 2004, *FleXPath* technique has been proposed by (Amer-Yahia et al., 2004). *FleXPath* depends on merging the XPath query language that has exact matching with full text search that has approximate matching. First, the query is converted to a tree and used it as a template to find the approximate matches within the XML document. The query relaxation process used by *FleXPath* depends on deleting a structural predicate if at least one of its nodes does not belong to the document structure and the deletion process will not affect the tree structure of the query. Furthermore, this technique performs more relaxation such as *contains-relaxation* by replacing the parameter of the *contains()* function with its ancestor, *tag-relaxation* by replacing a tag with its super tag, *value-relaxation,* and *type-relaxation*.

Instead of relaxing the query, (Lalmas and Rolleke, 2004) transforms the query to a conjunction query by adding an "OR" between the query's path and its predicate and changes each "AND" in the query with an "OR" to increase the recall precision. In this technique the XML document passes into two probabilistic transformation processes. In the first pass, each element, attribute name, attribute value and element value is attached with a probability value to indicate the importance of this element in the whole document using probabilistic object-oriented logic. The output from the first pass is transformed into probabilistic relational algebra expressions. This technique changes the XML document into a new one which is much bigger than the original.

In the same year, (Mandreoli et al., 2004) proposed a new approach for answering approximate queries to retrieve all the relevant parts of the XML document not only the exact matching. This approach finds the syntactic similarity between the XML Schema and the user's query, written in XQuery query language, and rewrites the query to match this Schema. It works with the XML Schema instead of working with the XML document directly in order to retrieve relative information from a repository of documents. Although this technique has the ability to retrieve 90% of the relevant information, it shows conflicts when the root node of the different schemas are the same as the root node in the query.

In 2006 (Li et al., 2006) proposed *FLUX* to process only range queries in their fuzzy appearance. It uses B+-tree in order to identify the relevant leaf nodes to the given user's query. The path from the root to the relevant leaves is

used as signatures to be matched with the path in the query to determine their relevancy. Using the Bloom filter, *FLUX* converts the path in the query and the path signatures into hash tables and compare between them to extract the most relevant paths. The implementation process for *FLUX* is limited only to two XML documents and the 100 tested queries include only the year and date range queries with random selections. Their test explains that *FLUX* perform good retrieval with higher speed that other relative techniques.

While *FLUX* tried to process fuzzy range queries, *TIJAH* (Mihajlovic et al., 2006) tried to process only two vague cases in NEXI query language. This technique finds the list of synonyms for each element name in the user's query using WordNet "A Lexical Database for the English Language", and it uses these synonyms as new keywords to be searched in the XML document by rewriting the query using the new elements. Furthermore, this technique generalizes the path in the query in order to look for the elements in the whole XML tree.

Depending on the aforementioned *FleXPath* approach, (Campi et al., 2009) proposed a new technique called *FuzzyXPath* that expands XPath query language to include fuzzy cases. The main purpose of this work was to determine the degree of similarity between two trees by providing a weight to each node to determine its importance within the document. The weight is calculated depending on the level of the node within the XML document and the number of its children. *FuzzyXPath* adds new functions to the list of available functions in XPath such as *SIMILAR* to find the similarity between the given node and the nodes in the document, and *CLOSE* to find the similarity between the given value and the data in the document. It provides more flexibility in path structure by adding *NEAR* and *BELOW* functions.

In our previous work (Al-Hamadani et al., 2009) we proposed a new technique to process vague queries by decomposing it into CAS and CO queries and then apply the normal retrieval process for each part. The results from the retrieval process are combined again to obtain the final results. The technique applied on health care record and it shows good retrieval precision.

(Fredrick and Dr.G.Radhamani, 2009) proposed a framework to extend XQuery language to include fuzzy queries. They tried to generalize the *FLWOR* to include natural language words, such as good, bad, etc. to get more precise

results. It depends on the fuzzy-set theory by (Zadeh, 1965) to transfer each fuzzy word to a range of values and then retrieve the most relevant parts from the document.

3.3 Problem Identification

The previous sections list several compression techniques that have the ability to process different kinds of queries. Table 3-4 lists all the discussed queriable compression techniques and shows the types of queries that can be processed by each technique. Some of the compressors require partial decompression to the compressed XML document in order to process some of these queries.

Table 3-4: Query types with the compression techniques process each.

Compression techniques/ query types	Simple queries (SQ)	Criteria queries (CQ)	Conjunctive queries (JQ)	Range queries (RQ)	Vague queries (VQ)
XGrind	☑	☑*	☑*	☑*	✘
Xpress	☑	☑	☑*	☑*	✘
XQZip	☑	☑	☑	☑*	✘
XQueC	☑	☑	☑	☑	✘
XSAQCT	☑	☑	✘	✘	✘
SXSI	☑	☑	☑	✘	✘

* Partial decompression required

It is clear that the entire existing compressors do not have the ability to process vague queries since this type of queries is complex and needs intensive research to resolve it.

For this reason, the research in this thesis is focused on how to handle different types of vague queries in retrieving information from compressed XML documents.

3.4 Chapter Summary

This chapter illustrated the main types of general purpose compressors and focused on XML compression techniques which rely on two types, either as queriable or non-queriable techniques. Since this research is dealing with a queriable compressor, this chapter concentrated on the existing techniques, listed their main features and the differences between them and the types of queries in the process. Finally, the chapter also demonstrated different techniques that have the ability to process vague queries and the key differences between them.

CHAPTER 4 XML Compressing and Vague Querying (XCVQ) Design

As shown in the literature review from the previous chapter, there are a good number of studies in the field of compressing XML documents and querying the compressed version without the need to fully decompress. However, vague queries, which are one of the most important query types, have been processed to retrieve information from raw XML documents and not from compressed ones.

Depending on the SDM as illustrated in Figure 1-1, the design of the complete system should be made, followed by its implementation which can be seen in Appendix-B. This chapter illustrates the design architecture of the *XCVQ* (an XML Compressing and Vague Querying) which has the ability to compress the XML documents and use the compressed files in order to retrieve information according to vague queries. It starts with the main architecture of the system followed by the design of each of its parts, namely *XCVQ*'s compressor, Decompressor, and the query processor.

4.1 System Architecture

As illustrated in Figure 4-1, the *XCVQ* system consists of two main stages. The first is designing a new XML compression technique which converts the normal XML documents to a compressed version. The second is designing a retrieving technique that processes the XPath vague queries in order to retrieve the relevant information from the compressed document accordingly.

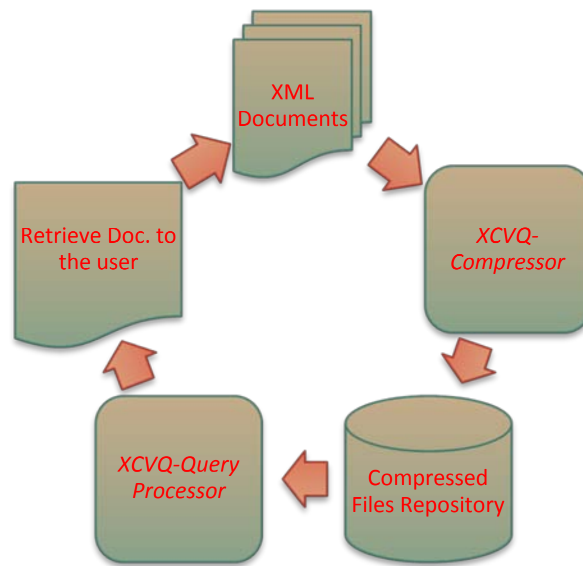


Figure 4-1: Preliminary Architecture of *XCVQ*

The design of the *XCVQ* does not rely on the XML Schema or the DTD of the document. This is due to several reasons:

1. The main purpose of designing *XCVQ* is to process vague queries which are usually written, as illustrated in a previous section, by inexperienced users who may not want to have another technology linked with their documents.
2. Even if the schema for a document exists, it could not have been accessible to the user.
3. Since the main purpose of any compressor is to reduce the storage memory and the transition bandwidth, *XCVQ* saves the amount of memory required to store the schema.

As illustrated in the design of the *XCVQ,* all the compressed XML documents are stored in a repository which is going to be used in the retrieving process. To the best of our knowledge, *XCVQ* may well be considered to be the first retrieving technique that has the ability to retrieve information from more than one XML document without requiring the pre-specification of the documents needed to be retrieved and without dependence on the document's schema. This

approach helps users retrieve more relative information no matter which documents contain this information. The complete design of the XCVQ is illustrated in Figure 4-2.

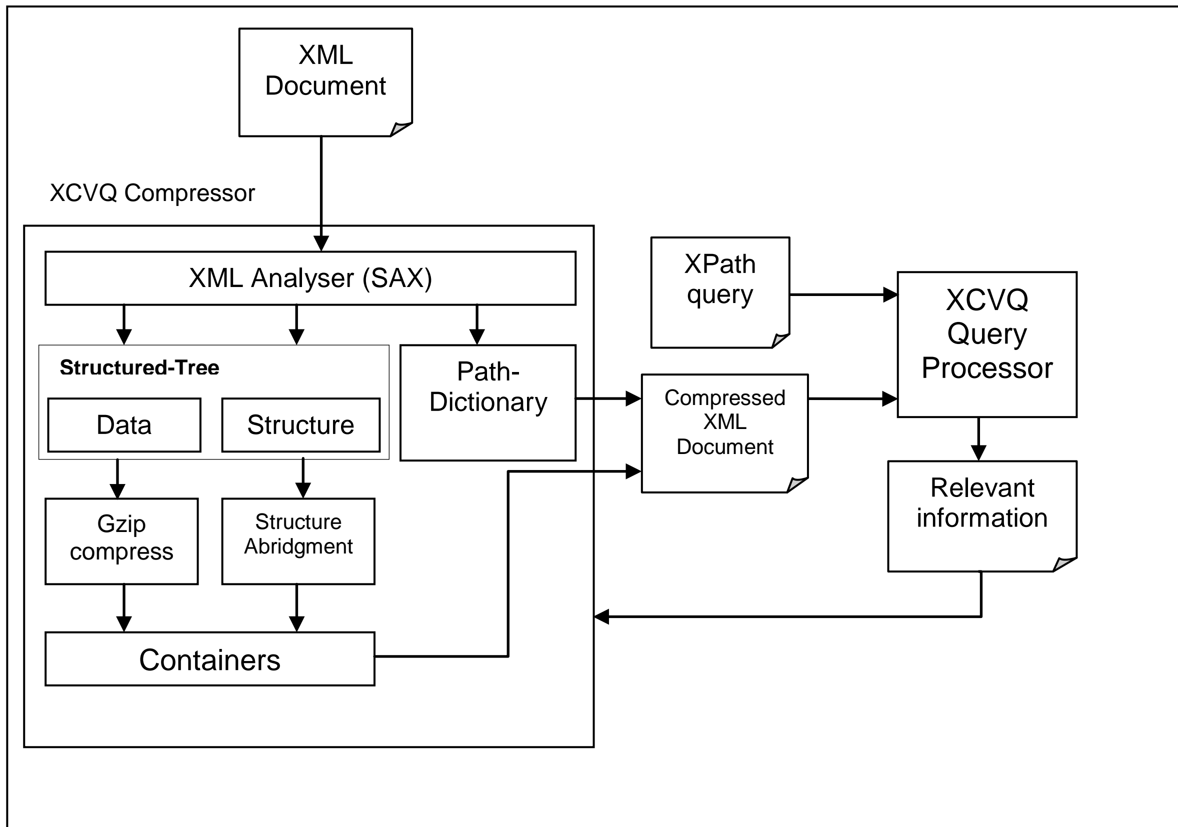


Figure 4-2: The complete design of XCVQ

The following sections demonstrate the design of each part of the system starting with *XCVQ-Compressor (XCVQ-C)*, passing by *XCVQ-Decompressor (XCVQ-D)*, and ending with *XCVQ-Query Processor (XCVQ-QP)*.

4.2 XCVQ-C Design

XCVQ-C compressor takes an XML document as the input and creates the compressed version from this document by passing through several steps. An

example in Figure 4-3 from (W3Schools.com, 2006b) will be used in the following sections in order to simplify the exact process of each step.

4.2.1 Creating the Structured-Tree & its Abridgment

As illustrated in Figure 4-2, the first step in compressing the XML document is to create the structured-tree using the SAX parser. This parser scans the XML documents only once and it cached several events such as start-document, start-element, end-element, and end-document. Section 2.2.2 contains more details about this parser and its advantages. During this parsing process the complete *path-dictionary* was created and separates the data part of the XML document from its structure to be abridged to the structured-tree. The structured-tree for the running example is shown in Figure 4-4. The data under each root-leaf path are stored in containers linked to that path.

```xml
<CATALOG>
    <CD no="1">
        <TITLE>Empire Burlesque</TITLE>
        <ARTIST>Bob Dylan</ARTIST>
        <COUNTRY>USA</COUNTRY>
        <PRICE>10.90</PRICE>
        <YEAR>1985</YEAR>
    </CD>
    <CD no="2">
        <TITLE>Hide your heart</TITLE>
        <ARTIST>Bonnie Tyler</ARTIST>
        <COUNTRY>UK</COUNTRY>
        <PRICE>9.90</PRICE>
        <YEAR>1988</YEAR>
    </CD>
    <CD no="3">
        <TITLE>Romanza</TITLE>
        <ARTIST>Andrea Bocelli</ARTIST>
        <COUNTRY>EU</COUNTRY>
        <PRICE>10.80</PRICE>
        <YEAR>1996</YEAR>
    </CD>
    <CD no="4">
        <TITLE>When a man loves a woman</TITLE>
        <ARTIST>Percy Sledge</ARTIST>
        <COUNTRY>USA</COUNTRY>
        <COMPANY>Atlantic</COMPANY>
        <PRICE>8.70</PRICE>
        <YEAR>1987</YEAR>
    </CD>
    <CD no="5">
        <TITLE>Black angel</TITLE>
        <ARTIST>Savage Rose</ARTIST>
        <COUNTRY>EU</COUNTRY>
        <PRICE>10.90</PRICE>
        <YEAR>1995</YEAR>
    </CD>
    <CD no="6">
        <TITLE>1999 Grammy Nominees</TITLE>
        <ARTIST>Many</ARTIST>
        <COUNTRY>USA</COUNTRY>
        <PRICE>10.20</PRICE>
        <YEAR>1999</YEAR>
    </CD>
</CATALOG>
```

Figure 4-3: An XML example

Each data item is accompanied with a number ID_{order} that represents the order of this item within the document (the number between the brackets in Figure 4-4). ID_{order} counts each start element, data value, attribute name, attribute value and end element. According to this number, each node is uniquely identified for the purposes of decompression process and in the querying process. Previous XML compressors used two numbers for each node in the structured-tree to identify this node uniquely. These numbers represented by the pre-order and post-order traversal of that node $[ID_{pre}, ID_{post}]$ (Cheng and NG, 2004; Arion et al., 2007; Arroyuelo et al., 2010) which required:

$$S_{SIT} = 2(N * \log_2 N) \quad (5)$$

Where S_{SIT} represents the number of bits needed to store the structured-tree that contains N nodes. While the number of bits required to store the same structured-tree in *XCVQ-C* is shown in equation (6).

$$S_{SIT} = (N * \log_2 N) \quad (6)$$

Since *XCVQ-C* uses only one number to store a node, the number of bits required to store a single node is $\log_2 N$. In this stage *XCVQ-C* saves half the number of bits required to store the structured-tree.

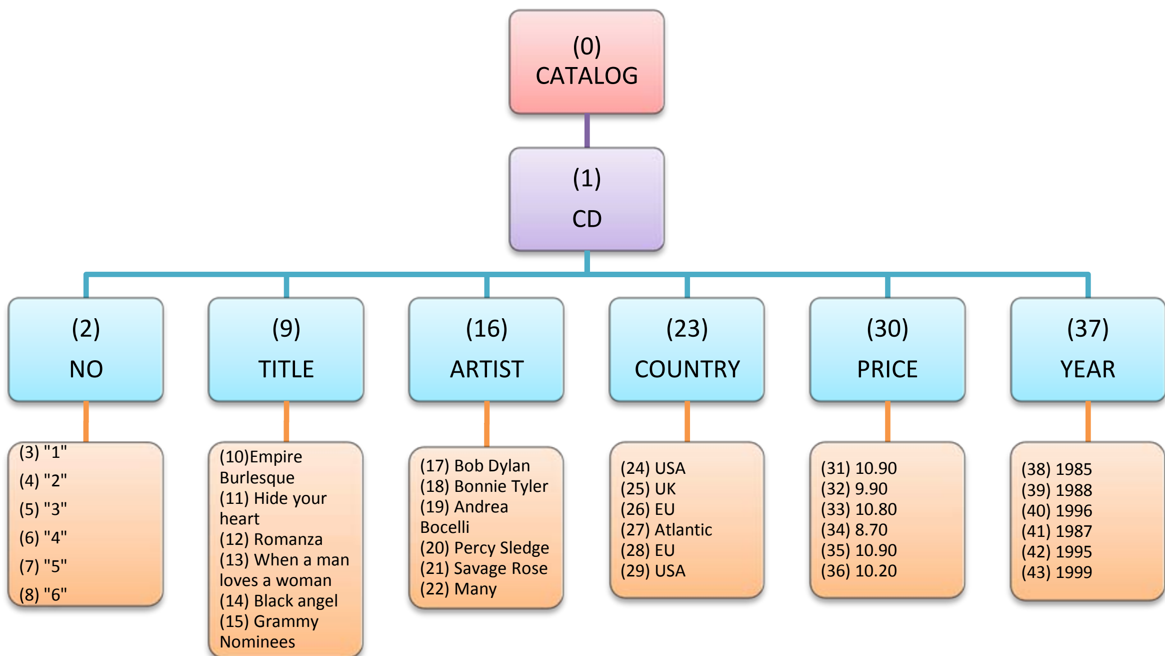


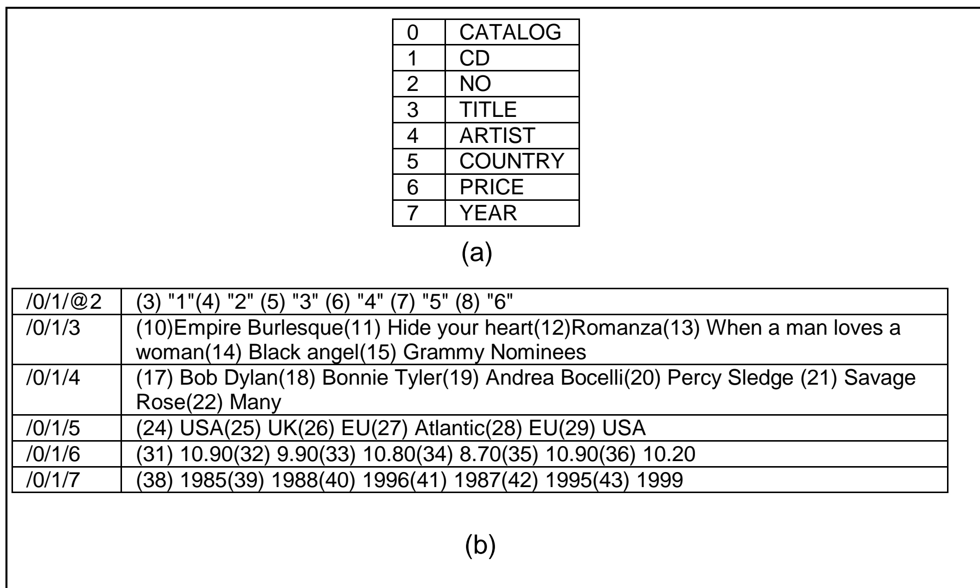
Figure 4-4: The structured-tree for the example in Figure 4-3

4.2.2 Creating the Containers

XCVQ-C creates the containers from the structured tree, as seen in Figure 4-2. First each node is replaced with a number that represents the entry of that

node's name in the *path-dictionary*. The structured-tree is traversed to create the containers. Each container has an index and data set. The path from the root to a leaf is used as index to the container and all the data under that path are the data set to this container.

For the running example, the *path-dictionary* and the containers are illustrated as in Figure 4-5, (a) and (b) respectively.



0	CATALOG
1	CD
2	NO
3	TITLE
4	ARTIST
5	COUNTRY
6	PRICE
7	YEAR

(a)

/0/1/@2	(3) "1"(4) "2" (5) "3" (6) "4" (7) "5" (8) "6"
/0/1/3	(10)Empire Burlesque(11) Hide your heart(12)Romanza(13) When a man loves a woman(14) Black angel(15) Grammy Nominees
/0/1/4	(17) Bob Dylan(18) Bonnie Tyler(19) Andrea Bocelli(20) Percy Sledge (21) Savage Rose(22) Many
/0/1/5	(24) USA(25) UK(26) EU(27) Atlantic(28) EU(29) USA
/0/1/6	(31) 10.90(32) 9.90(33) 10.80(34) 8.70(35) 10.90(36) 10.20
/0/1/7	(38) 1985(39) 1988(40) 1996(41) 1987(42) 1995(43) 1999

(b)

Figure 4-5: Creating containers process. (a) the path-Dictionary. (b) the container.

4.2.3 Compressing the Containers

After preparing all the containers and replacing the element's names in the containers with their entry in the *pathDictionary*, now the contents of the containers should be compressed using a back-end compressor. To do this, *XCVQ-C* uses two compressors to make comparison between them, LZW and GZIP compressors.

The granularity used by *XCVQ-C* is container/path, which means that after all the data parts of the document are settled in their appropriate containers, the back-end compressor is applied to compress each one of the containers separately. The decision made to choose this granularity is towards achieving a

balance between the compression ratio and the decompression process required. When dealing with back-end compressors, the higher the amount of data, the better is the compression ratio achieved. At the same time, this amount of data should not be the entire data part of the document, since they need to be decompressed in order to answer queries concerning them, so the technique needs to minimize the amount of data being decompressed. The previous XML compression techniques used different granularities to compress the XML documents using one of the back-end compressors, as shown in Table 4-1.

Table 4-1: Compression granularity comparison.

XML compression technique	Compression granules
XGrind	Value/tag
XPress	Value/path
XQzip	Blocks
XQueC	Container item/tag
XSAQCT	Container/tree-structure
SXSI	-
XCVQ	Container/path

- LZW Compression Technique

This is one of the dictionary-based lossless compressors which developed in 1984 from LZ78 by Lempel, Ziv and Welch (Salomon, 2007). It has been used in UNIX as a program compressor in 1986 and it is still being used by GIF, TIFF and PDF files to compress images (Murray and VanRyper, 1996). The tokens in LZW are pointers to their entries in the dictionary which starts with the first 256 positions occupied by the first 256 ASCII characters before any other entry.

Although it performs good compression ratio it suffers from problems. All the pointers to the dictionary should be larger than 8-bit since the first 256

entries are occupied from the beginning. This makes these pointers to be at least 3-bytes to accommodate all the entries in a document. Moreover, this technique is considered to be slow since its progress is one character at a time.

- Gzip Compression Technique

This is another example of dictionary-based lossless compression software which is based on *Deflate* compression algorithm. This software used by many applications such as HTTP protocol, the PNG (Portable Network Graphics), PNG images, and PDF files. The *Deflate* algorithm was designed in 2003 by combining the LZ77 and Huffman algorithms (PKWare, 2003; Salomon, 2007). *Deflate* uses different block sizes in order to compress the input data. The size of the blocks is determined according to the available memory and the size of the data. This algorithm provides three modes for each block, (1) No compression when the file is already compressed or it is random; (2) A fast mode that uses two fixed code tables in the encoder and they will not been written in the compressed file; and (3) A powerful mode that uses several code tables generated by the encoder and they should be written in the compressed file.

4.3 XCVQ-C Algorithms and Their Correctness

Since putting the complete compressing process in one algorithm could not be very clear, the designed algorithm is separated into three sub-algorithms. The separation process is made depending on the main parts of the XML document: start-element or attribute name, end-element, data, and end-document. This section illustrates the design of the algorithms in each of the previous XML parts and the formal correctness proof of each one of them. The process of correctness proof depends on specifying the set of preconditions $P:\{P_1, P_2, ..., P_n\}$ and the postconditions $Q:\{Q_1, Q_2, ..., Q_n\}$ and the algorithm A such that $P \xrightarrow{A} Q$ [ref]. The algorithm is considered to be true is it terminates and all the postconditions are true upon completion.

```
1.    Algorithm startElement(String eName, String data)
2.    let pathDictionary=[i0, i1, ..., in]
3.    let structured-Tree=[j0, j1, ..., jm]
4.    let pathStack=[kp, kp-1, ..., k0]
5.    let IDOrder= the current order
6.    if (eName ∉ pathDictionary)
7.          pathDictionary=[i0, i1, ..., in] ∪ eNamen+1
8.          Q″= n+1
9.    else
10.         Q″= q where iq =eName
11.   pathStack=[kp, kp-1, ..., k0] ∪ [Q″p+1]
12.   currentPath ← k0 + k1 + ...+ kp
13.   if (currentPath ∉ structured-Tree)
14.         Add currentPath to structured-Tree
15.   IDOrder++
16.   if (data is not empty)
17.         Add (IDOrder,data) to the leaf node of [j0, j1,
            ..., jm]
18.   End.
```

Figure 4-6: (*startElement)* algorithm

4.3.1 *startElement* algorithm

This algorithm in Figure 4-6 is processed whenever a start element or an attribute name occurs in the XML document. In this algorithm, each element or attribute name (*eName*) encountered in the XML document must be added to the list of *path-dictionary* if it is not added before (lines 6-8). The index of (*eName*) in the *path-dictionary* is used from now on instead of the element's name itself to be added to the structured-tree (lines 11-14). When there is a value in *data* this means that the algorithm is dealing with an attribute. In this case, the attribute value alongside with its order is added to the leaf of the current path in the *Structured-Tree* (lines 16-17).

To proof the formal correctness of this algorithm, the preconditions and the postconditions should first be specified:

P: { eName=a, data=b are two strings,

pathDictionary=[i_0, i_1, ..., i_n]=c represents the current *pathDictionary,*

structured-Tree=[j_0, j_1, ..., j_m]=d represents the current *structure-tree,*

pathStack=[k_p, k_{p-1}, ..., k_0]=e represents the current path elements stored in a stack}

Q: {$a \in c$,

$a \in e$,

$e \in d$,

In case of attributes, $b \in d$}

Correctness:

```
{eName=a, data=b, pathDictionary=c, structured-Tree=d, pathStack=e}
if (eName ∉ pathDictionary)
      {a∉c, data=b, pathDictionary=c, structured-Tree=d, pathStack=e}
      pathDictionary=[i0, i1, ..., in] ∪ eName(n+1)
      Q″= n+1
      {a ∈ c, data=b, pathDictionary=c, structured-Tree=d, pathStack=e}
else
      Q″= q where iq =eName
      {a ∈ c, data=b, pathDictionary=c, structured-Tree=d, pathStack=e}

pathStack=[kp, kp-1, ..., k0] ∪ [Q″(p+1)]
{a ∈ c, data=b, structured-Tree=d, a ∈ e}

currentPath ← k0 + k1 + ...+ kp
if (currentPath ⊈ structured-Tree)
      Add currentPath to structured-Tree
      {a ∈ c, data=b, e ∈ d, a ∈ e}
IDOrder++
if (data is not empty)
      Add (IDOrder,data) to the leaf node of [j0, j1,
..., jm]
     {a ∈ c, b ∈ d, e ∈ d, a ∈ e}=Q
```

```
1.  Algorithm endElement(String eName, String data)
2.     let pathStack=[k_p, k_p-1, ..., k_0]
3.     let structured-Tree=[j_0, j_1, ..., j_m]
4.     If data ≠ null
5.        Add (IDOrder,data) to the leaf node of [k_p, k_p-1, ...,
    k_0]
```

Figure 4-7: *(endElement)* algorithm

4.3.2 *endElement* algorithm

The algorithm in Figure 4-7 is processed when the end of an XML element encountered which means that there is a piece of data ready to be inserted in a leaf node of the structured-tree (if that element holds data). The suitable current path can be known from the contents of the *pathStack* and the *data* should be added in the leaf node of that path.

P: { eName=a, data=b are two strings,

pathStack=[k_p, k_{p-1}, ..., k_0]=c represents the current path elements stored in a stack,

structured-Tree=[j_0, j_1, ..., j_m]=d represent the current *structured-tree*}

Q: {*b* ∈ *d*}

Correctness:

{eName=a, data=b, *pathStack*=c, *structured-Tree*=d}
If *data* ≠ null
    {eName=a, data=b, *pathStack*=c, *structured-Tree*=d}
    Add (*IDOrder*,*data*) to the leaf node of [k_p, k_{p-1}, ..., k_0]
    { *b* ∈ *d* }= *Q*

```
1.    Algorithm endDocument ()
2.        let pathDictionary=[i0, i1, ..., in]
3.        let structured-Tree=[j0, j1, ..., jm]
4.        let F be the compressed file = Ø
5.        Add pathDictionary to F
6.        For all the N branches in structured-Tree
7.            index= Collect all the nodes [j0, j1, ..., jk]
8.            data= the contents of the leaf node for the path
      [j0, j1, ..., jk]
9.            data = GZipCompress(data)
10.           Add a Container(index, data) to F
11.   End.
```

Figure 4-8: *(endDocument)* algorithm

4.3.3 *endDocument* algorithm

When the whole XML document traversed, the algorithm in Figure 4-8 is processed. First the complete pathDictionary should be added to the output compressed file (line 7). The second step is to create the containers from the structured-tree and fill them with the compressed data (lines 6-10).

P : { *pathDictionary*=$[i_0, i_1, ..., i_n]$=a represent the complete *pathDictionary,*
structured-Tree=$[j_0, j_1, ..., j_m]$=b represent the current *structure-tree,*
$F = \emptyset$ }

Q: {$a \in F$, N Containers $\in F$ }
Correctness:

```
{PathDictionary=a, structured-Tree=b F=Ø }
   Add pathDictionary to F
   {a ∈ F, structured-Tree=b }
   For all the N branches in structured-Tree
```

```
        {a ∈ F, structured-Tree=b }
        index= Collect all the nodes [j0, j1, ..., jk]
        {a ∈ F, structured-Tree=b, index=path nodes}
       data= the contents of the leaf node for the path [j0, j1,
       ..., jk]
        data = GZipCompress(data)
       {a ∈ F, structured-Tree=b, index=path nodes, data is
compressed}
       Add a Container(index, data) to F
       {a ∈ F, structured-Tree=b, a container∈ F}
 end
 {a ∈ F, structured-Tree=b, N Containers ∈ F }=Q
```

4.4 XCVQ-D Design

As shown in Figure 4-6, to decompress the compressed XML file, *XCVQ-D* first applies the back-end decompression technique, either LZW or Gzip, to decompress only the contents of all the containers in order to get the data shown in Figure 4-5 for the running example.

The second step is to reconstruct the XML document from the indexes and the contents of the containers, and the *path-dictionary*. The main operation here is to determine the order of each element, attribute, and data value within the XML document. This order is the ID_{order} which is accompanied with the data in the containers but it should be checked against the number of data items written in the decompressed XML document ($\boldsymbol{D'}$) so far. To check the consistency of the order, *XCVQ-D* uses equation (7) such that:

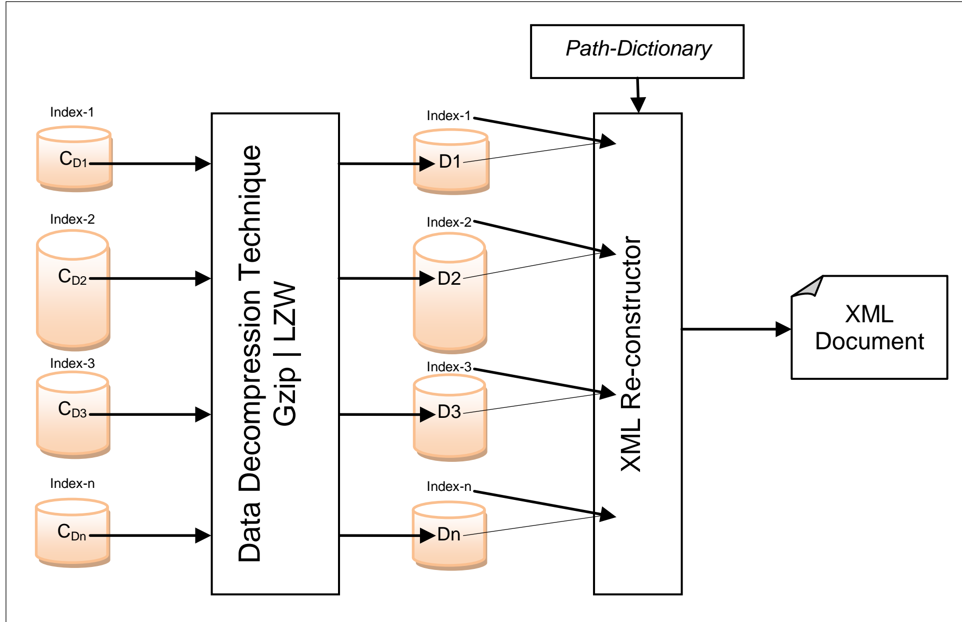


Figure 4-9: Architecture of *XCVQ-D*

Definition-1: If *XCVQ-D* has a piece of data [($\boldsymbol{\sigma}$)d], where $\boldsymbol{\sigma}$ denotes the ID_{order} accompanied with the data in a container, $\boldsymbol{\sigma'}$ is the ID'_{order} which denotes the order of the data written so far in $(\boldsymbol{D'})$, then the new order of D should be calculated by getting the difference between ($\boldsymbol{\sigma}$) and the ($\boldsymbol{\sigma'}$) taking into consideration the number of the elements and attribute names still not written in $(\boldsymbol{D'})$, such that:

$$\boldsymbol{C_{order}} = \boldsymbol{\sigma} - (\boldsymbol{\sigma'} + [\boldsymbol{\mu_P} - \boldsymbol{\mu_C}]) \quad (7)$$

Where:

$\boldsymbol{\mu_P}$*: The number of elements and attribute names written in* $\boldsymbol{D'}$*.*

$\boldsymbol{\mu_C}$*: The number of elements and attribute names in the index of the container having this data.*

Then the value of C_{order} is checked and a performance made as shown in equation(8).

$$\boldsymbol{if\ C_{order}} \begin{cases} = \boldsymbol{0} & \boldsymbol{newID_{order}} = \boldsymbol{\sigma} \\ \neq \boldsymbol{0} & \boldsymbol{D'} = \boldsymbol{D'} \cup ([\boldsymbol{\mu_P}] - [\boldsymbol{\mu_C}]) \end{cases} \quad (8)$$

If C_{order} equals to (0) this means that the current ID_{order} is consistent with the number of elements and attribute names in $\boldsymbol{D}'$. Otherwise, the difference between the current path in $\boldsymbol{D}'$ and the index path for the current container should be added to $\boldsymbol{D}'$ before adding the required data.

■

Since the decompression method depends on the existence of data, the resulted decompressed XML document is lossless from the data side while the dummy nodes (the element that has no data but consists of an open tag and a close tag) could be lost. This case appears in documents that are converted from a database system with poorly structured documents.

4.5 XCVQ-D Algorithm and its Correctness

The algorithm in Figure 4-10 illustrates the process of *XCVQ-D* which takes the *pathDictionary* and the compressed containers as its parameter list. The main idea of the decompression algorithm is to look for a data value which has the minimum ID_{Order} and put it in the decompressed file in its appropriate place. From the design of the *XCVQ-C* the first data value in each container always has the minimum order within this container, the process of looking for the minimum order will check only *(n)* item, where *(n)* represents the number of containers instead of searching all the data in the containers. This process reduces the time required to decompress the containers to $O(n)$ instead of $O(n \times m)$ where m represents the number of data items in each container.

If this piece of data is the first data value in the XML document (line 9) then, all the path's elements in the index of the container holding this data are pushed in a stack which represent the current working path and add these elements (or attribute names) to the new XML document ($\boldsymbol{\acute{D}}$) as an open tags (lines 10-12).

Before adding the data to the output file, a consistency check is made, by following the instructions in lines 16-19, where ({}) means set difference between the contents of the stack and the index path. If there is no consistency (line 17) then the difference between the stack content and the index path is added to the output file as losing tags and then the piece of data is added to the output file. In every addition to the output file, the value of (*dataOrder*) is updated (lines 12, 19, and 21) to check the consistency between them each time.

After adding the data to the output file, this data alongside with its order is deleted from the container. This process is done to keep the order of the first data items in all the containers in their minimum values and to release the memory storage used by these data values.

This process is continued until all the containers are empty, then all the content of the stack is added to the output file as closed tags to finish the new decompressed XML document.

```
1.  Algorithm XCVQ-D (pathDictionary [P0, P1, ...,Pm], containers
    [C0, C1, ...,Cn])
2.  let currentPathStack=[Sk, Sk-1, ...,S0]
3.  let dataOrder= number of elements, attribute names, and data
    values in the outputFile D́
4.  While (the containers are still having data) Do
5.   for i=0 to n
6.       let minDataSet[(O0,D0), (O1,D1),... (On,Dn)] ← first
         element in each container
7.       minOrder=min(O0, O1,..., On)
8.       minData ← Di from (minOrder, Di)
9.     if currentPathStack=∅
10.      currentPathStack ← Ci.index
11.      D́ ← open tags of Ci.index
12.      dataOrder=dataOrder+ number of open tags added
13.     }
14.    Else
15.      currentPathStack ← [Ci.index]-[currentPathStack]
16.    dataCons= minOrder − (dataOrder + {[Ci.index] − [currentStackPath]})
17.    if dataCons ≠ 0
18.      D́ ← close tag {[Ci.index] − [currentStackPath]}
19.     dataOrder=dataOrder+ number of close tags added
20.    D́ ← Di
21.    dataOrder=dataOrder+1
22.    remove (minOrder, Di)
23. }
```

Figure 4-10: *XCVQ*-Decompression algorithm

The next paragraph discuss the correctness of the decompression algorithm by guarantee the one-to-one mapping from the compressed document to the decompressed document.

The core of the decompression technique is to make sure that each part of the compressed XML document should return to its place in the original XML document. This is done in *XCVQ* by using the ID_{Order} which counts the order of each single part of the original document, such that:

$$O_{data}(ID_{Order})=D_{data}(ID_{Order})$$

Where, O_{data} and D_{data} represent a single piece of data in the original and the decompressed XML documents, respectively.

As seen in Figure 4-5, the only $O_{data}(ID_{Order})$ stored in the compressed document are for the data part of the document to save the storage required. For instance, in the first container indexed *(/0/1/@2)* in Figure 4-5, the first data item ("1") has its $O_{data}(ID_{Order})$=3. This means that there are three pieces should be transferred to the decompressed XML document before transferring this part of data. These pieces are the three nodes in the container's index *(/0, /1, and /@2)*.

To make this balance between $O_{data}(ID_{Order})$ and $D_{data}(ID_{Order})$ in the decompression algorithm, the *dataOrder* variable was used (to represent $D_{data}(ID_{Order})$) to count every single piece of data written in the XML document. Before adding a data value to the decompressed XML document, the decompression algorithm checks if it is in its right place (i.e. if $O_{data}(ID_{Order})=D_{data}(ID_{Order})$ after taking onto consideration the expected number of pieces from the). Otherwise a process is required to solve this inconsistency between the two values and as follows:

1. Find the difference between the two ID_{Order}s

   $D=O_{data}(ID_{Order}) - D_{data}(ID_{Order})$

This means that there are D pieces should be added to the decompressed file first.

2. Add the D pieces of data that are in the current working path but not in the container's index.
3. Update $D_{data}(ID_{Order})$

   $D_{data}(ID_{Order}) = D_{data}(ID_{Order}) + D$

4. *Since*

   $D = O_{data}(ID_{Order}) - D_{data}(ID_{Order})$

   Then

   $D_{data}(ID_{Order}) = D_{data}(ID_{Order}) + O_{data}(ID_{Order}) - D_{data}(ID_{Order})$

$D_{data}(ID_{Order}) = O_{data}(ID_{Order})$ Which is the target of the decompression technique.

4.6 XCVQ-QP Design

The design of the query processor, as illustrated in Figure 4-7, consists of various stages. The output(s) from each stage is used as input for the other stages. The role of each stage and its design are discussed in the next sections using the same running example in Figure 4-3.

4.6.1 XPath Query

The current XPath query language does not have the ability to answer vague queries, since its work is based on a restricted Boolean matching; either the query matches part(s) of the existing document and retrieves those parts, or no retrieval at all is achieved if there is no match. *XCVQ-QP* uses XPath as a query language after expanding the original language to give it the ability to solve vague user's queries. This expansion includes adding more flexibility in both path matching and data value matching in addition to adding some functions to the list of available XPath functions.

- Path Matching Expansion

To increase the flexibility of XPath axes matching, *XCVQ-QP* provides some generalization to the XPath query that gives the users of *XCVQ-QP* the ability to retrieve the most relevant information to their queries (Grust, 2002; Amer-Yahia et al., 2004; Campi et al., 2009). These generalizations are the following:

1. Eliminating the use of the recursive descent sign (//) and replacing it with the child operator (/) sign. This elimination increases the flexibility of *XCVQ* as shown in the following examples:

    Example (1): to retrieve all the (*TITLE*) elements from the XML example in Figure 4-3, an XPath query should be `(/CATALOG/CD/TITLE)`. In this case the user should have a complete idea about the XML schema for that file to indicate the complete path from the root to the (*TITLE*) element. To retrieve the same information, *XCVQ* query is either `(/CD/TITLE)` or `(CATALOG/TITLE)`, which is simpler than XPath queries and does not need any previous knowledge about the schema.

    Example (2): if the user need the (*TITLE*) element for the *CD* with (*no*) equals to "2". The XPath query is `(CATALOG/CD[@no="2"]/TITLE)` while the *XCVQ* query is `(/CD[@no="2"]/TITLE)` which is again much simpler than XPath query.

2. If the query tries to retrieve sibling elements, then using XPath would need to write two separate queries or one query with two parts connected by logical (and) operator.

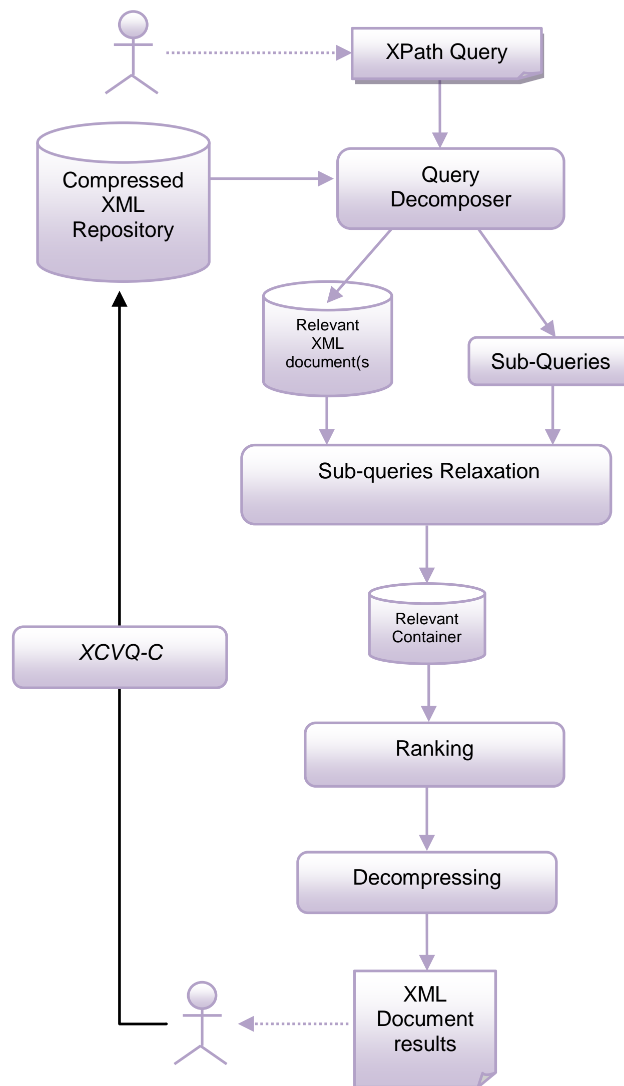


Figure 4-11: The architecture of the query processor.

Example (3): The elements (*TITLE*) and the elements (*YEAR*) are both siblings in the XML tree. The XPath query to retrieve all the data from the two elements is `(CATALOG/CD/TITLE | CATALOG/CD/YEAR)`, while the *XCVQ* query to retrieve the same information is `(/TITLE/YEAR)`.

Example (4): if the user interesting in retrieving all the (*TITLE*) elements only for the *CD* published after (*1990*). The XPath

query is `(/CATALOG/CD/TITLE and /CATALOG/CD[YEAR>1990])`, while the relevant *XCVQ* query is `(/TITLE[YEAR>1990])`.

3. If the order of the path is not arranged properly, XPath query does not have the ability to retrieve information from the specified document, while *XCVQ* does.

   Example (5): using the same requirements of *Example (3)*, XPath user should follow the same path from the root to the required element, while *XCVQ* query could be written as follows: `(TITLE/CD/YEAR)`

4. Using XPath queries, the user should follow the case of the letters, since the XPath query is case sensitive language. This feature adds more complexity to the user and to the XML creator who has to follow those specific rules. *XCVQ* queries are case insensitive, which retrieve the information from the XML document even if the case is different.

   Example (6): all the *XCVQ* queries in *Examples (1-5)* can be written as following:

```
cd/TITLE
/CD[@no="2"]/title
/TITLE/year
title[year gt 1990]
title/cd/YEAR
```

5. When the system does not find a specific element within the XML compressed database, it tries to look for elements that are similar to it. For that reason, *XCVQ-QP* uses a *string-similarity* algorithm (White, 2008) in order to match any misspelling in the elements or attribute names. If an element within the path is written in a wrong way, then the system will look for the nearest spelling element in the retrieved documents such that the similarity ratio should not be below 40%. After many experiments, we noticed that this percent is the best for retrieving the required element. If this number is less than 40, then non-related elements could be retrieved.

— The choice of *string-similarity* algorithm is made on the ground that it meets most of *XCVQ*-QP needs since this algorithm has the following features:

1. If two strings have minor differences, they are considered to be similar (ex: heap, heard).
2. If two strings have the same words but in different order, they are considered to be similar (ex: data base management system, managing data base).

As shown in Figure 4-12, the *string-similarity* match algorithm takes two strings, and for each string it produces sets each of which has two adjacent letters in that string. Then the similarity is computed as in line (5) to determine the similarity ratio between them.

```
1. Algorithm string-similarity (String st1, st2)
2. let st1= [s1s2s3 ... sn]   and st2= [c1c2c3 ... cm]
3. st1Set= [s1s2][s2s3][s3s4]...[sn-1sn]
4. st2Set= [c1c2][c2c3][c3c4]...[cm-1cm]
5. SimilarityRatio = 2×|st1Set∩st2Set| / (|st1Set|+|st2Set|)
6. End.
```

$$SimilarityRatio = \frac{2\times|st1Set \cap st2Set|}{|st1Set|+|st2Set|}$$

Figure 4-12: S*tring-similarity* match algorithm

6. *XCVQ-QP* has the ability to retrieve information from more than one file (FAZZINGA et al., 2009) even if the user does not specify these files in prior. As a simple example, if the user has the query (title/year) then s/he might get information about the titles and year of publication for CDs, movies, books or journals. Moreover, *XCVQ-QP* has the ability to compare the results from one file with the data from another file and retrieve the results accordingly.

Example (7): Suppose the following user query: `(catalog/book/author/bookstore[author="Erik Ray"])`. This query is considered to be a merged from two queries, `(catalog/book/author="Erik Ray")` and `(bookstore/book/author="Erik`

`Ray")`. Each one is to retrieve all the books for the author's name `("Erik Ray")` from two separate XML documents that follow different schemas.

Example (8): suppose the following user query:
`(cars/car/price lt carType/Ford[model eq 2008]/price)`
In this query the user is interesting in looking for all the cars that their pricees are less than (*lt*) the 2009 Ford car price. *XCVQ-QP* first looks for the smallest price (*x*) in the path `(carType/Ford[model eq 2008]/price)` and then retrieve all the information from the path `(cars/car/price)` which are less than (*x*). Notice that the two paths are from two separate XML documents.

- Data Value Matching Expansion

In order to make more expansion on XPath queries to retrieve more relevant data from the XML document, *XCVQ-QP* adds a set of functions that deal with the data part of the document (Campi et al., 2009). Although some of these functions are adopted to be used in structural retrieval as well. The extended functions with examples of their use are illustrated in the next section.

1. *Synonym(x):* This function is created to be used to retrieve information from both the structure and the data parts. It has only one parameter *x* and returns a list of synonyms for *x.*

   To do so, *XCVQ-QP* uses the WinterTree thesauruses engine (WinterTree, 2006)which provides a wide multipurpose dictionary. This engine provides the user the ability to modify its dictionary by adding new words with their synonyms or adding more synonyms to the existing words.

   If the list of synonyms is (S_1, S_2, …, S_n), then *XCVQ-QP* processes (n) queries by replacing each Si instead of the function call.

   Example (9): Suppose the query `/cd/title/synonyms("time")`. *XCVQ-QP* replaces this query with 3 queries:

```
/cd/title/duration
/cd/title/interval
/cd/title/date
```

It is clear that the use of the function in this example is for structure retrieving purposes.

Example (10): Suppose the query `/cd/country eq synonyms("Britain")`, which retrieves information from the data part of the XML document. This query is replaced with two queries:

```
/cd/country eq "UK"
/cd/country eq "The United Kingdom"
```

2. *Similar(x):* This function uses the String-Similarity algorithm, shown in Figure 4-8 in order to retrieve information according to one of the following conditions:
   a) If the user has doubt on the spelling of a string as shown in *Example (11)*.
   b) If the similar strings to *x* are required as shown in *Example (12)*. This function works on both the structure and the data parts of the XML documents depending on the previous conditions.

Example (11): In the query `/cd/title/similar(artest)`, the word `artest` has spelling error. The role of *XCVQ-QP* here is to find the similar element name from the retrieved documents and retrieve the required information accordingly. This query is replaced with `/cd/title/artist` for the running example.

Example (12): In the query `/cd/year/title eq similar("keep your heart")` for the running example, the data required is similar to `"keep your heart"` which is replaced by *XCVQ-QP* with `cd/year/title eq "hide your heart"`.

- Function Set Expansion

The list of available functions in XPath query language includes string, Boolean and number functions. *XCVQ-QP* adds four functions to the number functions set.

1. *Average(x):* the *avg()* function in XPath provides the user the ability to get the average of a list of numbers specified as a parameter list for the function. This function is expanded by *XCVQ-QP* to provide the user the ability to specify an element from the XML document and find the average of the numerical values under that element.

   Example (13): The query /cd/title/average(price) retrieves all the data values of the title element and the average of the numbers of the price element.

2. *Median(x):* This function is used to find the median for the list of numbers in the selected path.

   Example (14): If the query `/catalog/cd median(year)`is applied, the user will get the median number for all the data values of the `year` element.

3. *Between(x,y):* Instead of using and logical operator to retrieve information lying between two different intervals, *XCVQ-QP* introduce this function. It has two parameters which represent the data interval.

   Example (15): The query `/cd/title[price=between(9.0,10.0)]` retrieves all the `title` elements if and only if the value of its `price` element is between the given interval.

4.6.2 Query Decomposer

This part of the *XCVQ-QP* is responsible for decomposing the XPath query into several sub-queries. This stage consists of two decomposition stages, as shown in Figure 4-9. Each stage has specific roles and results in a set of sub-queries as in the following:

Decomposition Stage -1: The main purpose of this stage is to specify the relevant documents from the compressed XML repository. This case occurs

when the user's query does not specify the exact XML document to retrieve information from it. *XCVQ-QP* decomposes this query into (n) queries, where (n) represents the number of the relevant documents.

Definition- 2 (relevant document): If $\boldsymbol{Q} = [\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \dots \boldsymbol{e_n}]$ is the set of elements in the user's query, and $\boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \dots \boldsymbol{x_m}]$ is the set of all the compressed XML documents in the repository. $\boldsymbol{x}' \in \boldsymbol{X}$ is considered to be relevant document if $\exists \boldsymbol{e_i}: \boldsymbol{e_i} \in \boldsymbol{x'_j}.\boldsymbol{PD}$, where $\boldsymbol{PD}$ is the *path-dictionary* for the specified document. If so, add $\boldsymbol{x'_j}$ to the relevant repository and add $\boldsymbol{e_i}$ to $\boldsymbol{q'_j}$.

■

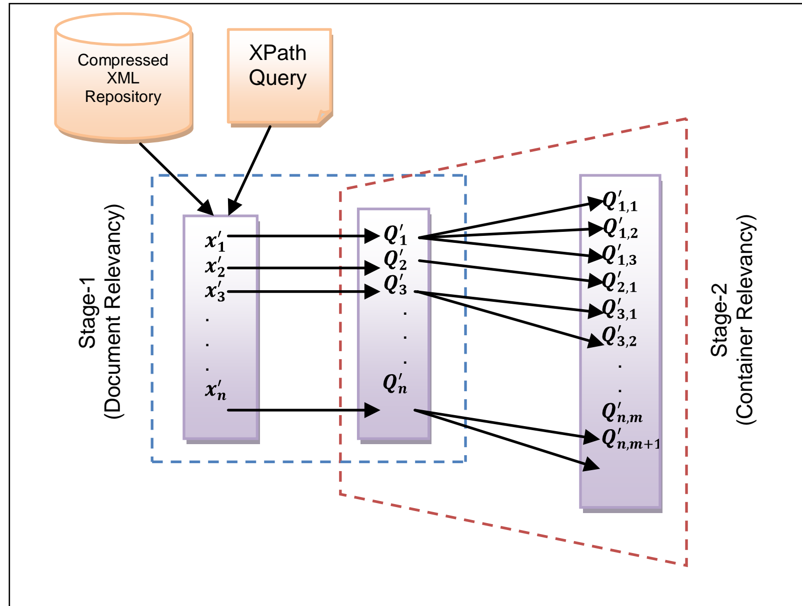


Figure 4-13: The design of *XCVQ*-Query Decomposer

According to the definition above, a XML document is considered to be relevant if it has one or more of the query elements in its path-dictionary. All the relevant documents $\boldsymbol{X}' = \boldsymbol{x'_1}, \boldsymbol{x'_2}, \dots \boldsymbol{x'_l}$ are collected in a small repository for relevant XML documents each of which is accompanied with its relevant sub-query $\boldsymbol{Q}' = \boldsymbol{q'_1}, \boldsymbol{q'_2}, \dots \boldsymbol{q'_l}$. All the elements and attribute names in $\boldsymbol{Q}'$ are replaced with its location in the path-dictionary of the relevant document. This process is done to prepare the sub-queries for the second decomposition stage. As an

example, if a sub-query is: `title/cd/year` from the running example, it is replaced with `/3/1/7` where `3, 1,` and `7` represent the entries of `title, cd,` and `year` respectively as shown in Figure 4-5.

In the case when the user's query $\boldsymbol{Q}$ is submitted to retrieve information from a specific document, then the query does not pass by this stage and the list of sub-queries $\boldsymbol{Q}'$ has only the original query, i.e. $\boldsymbol{Q}' = [\boldsymbol{Q}]$ and the related document's repository contains only the specified document.

Decomposition Stage -2: After specifying the relevant documents, the role of this stage is to specify the relevant containers within these documents. This process causes further decomposition to the sub-queries

> *Definition-3* (Relevant Container): Given $\boldsymbol{C} = \{\boldsymbol{c_1}, \boldsymbol{c_2}, \cdots \boldsymbol{c_n}\}$ *represents the set of n containers for a relative document and each of these containers has an index with* $\boldsymbol{k}$ *elements* $\boldsymbol{P} = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \cdots \boldsymbol{p_k}\}$, *and* $\boldsymbol{q'_i} = \{\boldsymbol{e_1}, \boldsymbol{e_2}, \cdots \boldsymbol{e_m}\}$ *represents the set of m elements in the sub-query accompanies C, to select the relevant container follow the steps:*
>
> $\forall \boldsymbol{c_i} \in \boldsymbol{C}$
>
> $\boldsymbol{k}$ = *the last element in the set*
>
> $\boldsymbol{if}\ \boldsymbol{p_k} \in \boldsymbol{q'_i}$
>
> *Add* $\boldsymbol{c_i}$ *to* $\boldsymbol{C}'$ *to denote the list of relevant containers*
>
> $\forall \boldsymbol{e_k}$ *if* $\boldsymbol{e_k} \in \boldsymbol{P}$, *copy* $\boldsymbol{e_k}$ *and add it to the list of the elements in* $\boldsymbol{q'_{i,j}}$ *to denote a new sub-query.*
>
> $\boldsymbol{k} = \boldsymbol{k}-1$
>
> *Repeat the above steps until the entire element in* $\boldsymbol{q'_i}$ *are copied to a new sub-query.*
>
> ■

At the end of this stage only the relevant containers taken from the relevant documents are uploaded into the memory for ranking process. Each of these containers is accompanied with is sub-query as shown in Figure 4-13.

4.6.3 Query relaxation

In this stage, the list of sub-queries $\boldsymbol{Q'}$ is being relaxed to determine the relevancy of each of these queries to the document. To do so, *XCVQ-QP* relaxes all the members of $\boldsymbol{Q'}$ according to each of the containers in $\boldsymbol{x'}$ to compute the cost of this relaxation process for ranking purposes. To reach this goal, *XCVQ-QP* adopts different kinds of relaxation processes. These types and their costs are listed below:

1. Node insertion:

   This type of relaxation is done by inserting one node or more in the list of available query nodes. To do so, *XCVQ-QP* compares each container of the relevant XML documents with its sub-query.

   Definition-4 (Node-Insertion): Given a *container* $\boldsymbol{c_j} \in \boldsymbol{C'}$ *has an index with* $\boldsymbol{k}$ *elements* $\boldsymbol{P} = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \cdots \boldsymbol{p_k}\}$ *and given* $\boldsymbol{q'_{i,j}} = \{\boldsymbol{e_1}, \boldsymbol{e_2}, \cdots \boldsymbol{e_m}\}$ *represents the set of m elements in the sub-query accompanies the* i^{th} *relevant document and* j^{th} *relevant container. The relaxed sub-query* $\boldsymbol{q''_{i,j}}=\boldsymbol{q'_{i,j}} \cup (\boldsymbol{P} \setminus \boldsymbol{q'_{i,j}})$

   ■

   The cost of the insertion node(s) in a single sub-query is specified as follows:

$$\boldsymbol{cost_{In}} = \frac{|(\boldsymbol{P} \setminus \boldsymbol{q'_{i,j}})|}{|\boldsymbol{P}|} \tag{9}$$

2. Node renaming:

   After completing the first stage of relaxation, each sub-query is going to pass through the following procedure:

   Let $\boldsymbol{q''_{i,j}} = \{\boldsymbol{e_1}, \boldsymbol{e_2}, \cdots \boldsymbol{e_m}\}$ be the set of elements in the current sub-query, $\boldsymbol{P} = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \cdots \boldsymbol{p_k}\}$ be the set

of elements of the index for the current container associated with $q''_{i,j}$.

For all $e_i \in q''_{i,j}$,

if $e_i \notin P$ then find the value of *SimilarityRatio* by applying the *string-similarity(*e_i, p_j) algorithm in Figure 4-8 such that $1 \le j \le m$.

if *(SimilarityRatio>50%)* then

replace e_i with p_j

changes++

The cost of the node renaming process is calculated as follows:

$$cost_{Ren} = \frac{changes}{|q''_{i,j}|} \tag{10}$$

3. Node deletion:

After inserting all the required nodes from the index of a container, the extra nodes from the query should be removed.

Definition-5 (Node-deletion): Given a *container* $c_j \in C'$ *has an index with* k *elements* $P = \{p_1, p_2, \cdots p_k\}$*and given* $q'_{i,j} = \{e_1, e_2, \cdots e_m\}$ *represents the set of m elements in the sub-query accompanies the i*[th] *relevant document and j*[th] *relevant container. The relaxed sub-query* $q''_{i,j} = q''_{i,j} - [q''_{i,j} \setminus P]$.

■

The cost required to delete node(s) from a sub-query is:

$$cost_{Del} = \frac{\left|q''_{i,j} \setminus P\right|}{\left|q''_{i,j}\right|} \tag{11}$$

The deletion cost of all the sub-queries will never be equal to 1 (which means all the elements in the query are deleted), since all these queries passed by the insertion relaxation first and all the irrelevant containers are dismissed.

4. Order relaxation:

   This is the last relaxation process which arranges the order of the nodes in each resulted sub-queries. The cost of this relaxation is shown in equation (11) such that *changes* represent the number of changing in the order of the elements in the sub-query.

$$cost_{Order} = \frac{changes}{|Q''|} \tag{12}$$

Example (16): For the running example, the lists of containers indexes are as follows:

$C_1 = \{CATALOG, CD, NO\}$
$C_2 = \{CATALOG, CD, TITLE\}$
$C_3 = \{CATALOG, CD, ARTIST\}$
$C_4 = \{CATALOG, CD, COUNTRY\}$
$C_5 = \{CATALOG, CD, PRICE\}$
$C_6 = \{CATALOG, CD, YEAR\}$

Suppose the following vague query:

```
Q1=document("cdcatalog.xml")/title/cd/artest[year between(1990,
1996)]
```

$Q = \{TITLE, CD, ARTIST, YEAR\}$

Since this query specifies the XML document to retrieve information from it, it is going to pass through stage-2 directly to specify the related containers. The following list illustrates the related containers alongside with the sub-query accompanied it:

$$\boldsymbol{C_2} = \{\boldsymbol{CATALOG, CD, TITLE}\} \Leftrightarrow \boldsymbol{q'_{1,2}} = \{\boldsymbol{TITLE, CD}\}$$

$$\boldsymbol{C_3} = \{\boldsymbol{CATALOG, CD, ARTIST}\} \Leftrightarrow \boldsymbol{q'_{1,3}} = \{\boldsymbol{CD, ARTEST}\}$$

$$\boldsymbol{C_6} = \{\boldsymbol{CATALOG, CD, YEAR}\} \Leftrightarrow \boldsymbol{q'_{1,6}} = \{\boldsymbol{CD, YEAR}\}$$

The insertion relaxation process updates the sub-queries to be as follows:

$$\boldsymbol{q''_{1,2}} = \{\boldsymbol{CATALOG, TITLE, CD}\}$$

$$\boldsymbol{q''_{1,3}} = \{\boldsymbol{CATALOG, CD, ARTEST}\}$$

$$\boldsymbol{q''_{1,6}} = \{\boldsymbol{CATALOG, CD, YEAR}\}$$

The cost of insertion the required nodes are as follows:

$$\boldsymbol{cost_{In_{q''_{1,2}}}} = \frac{|\{\boldsymbol{CATALOG, CD, TITLE}\} \setminus \{\boldsymbol{TITLE, CD}\}|}{|\{\boldsymbol{CATALOG, CD, TITLE}\}|} = \frac{\mathbf{1}}{\mathbf{3}}$$

$$\boldsymbol{cost_{In_{q''_{1,3}}}} = \frac{|\{\boldsymbol{CATALOG, CD, ARTIST}\} \setminus \{\boldsymbol{CD, ARTEST}\}|}{|\{\boldsymbol{CATALOG, CD, ARTIST}\}|} = \frac{\mathbf{1}}{\mathbf{3}}$$

$$\boldsymbol{cost_{In_{q''_{1,6}}}} = \frac{|\{\boldsymbol{CATALOG, CD, YEAR}\} \setminus \{\boldsymbol{CD, YEAR}\}|}{|\{\boldsymbol{CATALOG, CD, YEAR}\}|} = \frac{\mathbf{1}}{\mathbf{3}}$$

The node renaming relaxation process updates the sub-queries to be as follows:

$$\boldsymbol{q''_{1,2}} = \{\boldsymbol{CATALOG, TITLE, CD}\}$$

$$\boldsymbol{q''_{1,3}} = \{\boldsymbol{CATALOG, CD, ARTIST}\}$$

$$\boldsymbol{q''_{1,6}} = \{\boldsymbol{CATALOG, CD, YEAR}\}$$

The only sub-query affected by this stage is $\boldsymbol{q''_{1,3}}$ and the cost of this process is as follows:

$$\boldsymbol{cost_{Ren_{q''_{1,3}}}} = \frac{\mathbf{1}}{|\{\boldsymbol{CATALOG, CD, ARTEST}\}|} = \frac{\mathbf{1}}{\mathbf{3}}$$

The node reordering relaxation process updates the sub-queries to be as follows:

$\boldsymbol{q''_{1,2}} = \{\boldsymbol{CATALOG, CD, TITLE}\}$

$\boldsymbol{q'_{1,3}} = \{\boldsymbol{CATALOG, CD, ARTIST}\}$

$\boldsymbol{q'_{1,6}} = \{\boldsymbol{CATALOG, CD, YEAR}\}$

The only sub-query affected by this stage is $\boldsymbol{q''_{1,2}}$ and the cost of this process is as follows:

$$\boldsymbol{cost_{Order_{q''_{1,2}}}} = \frac{\mathbf{1}}{|\{\boldsymbol{CATALOG, TITLE, CD}\}|} = \frac{\mathbf{1}}{\mathbf{3}}$$

4.6.4 Ranking

After relaxing all the sub-queries, the process of finding the similarity between the containers' index and the sub-queries is computed according to the following equation:

> *Definition-6 (Query Similarity)*: To find the similarity between the given query (Q) and the relevant XML document, first the cost of all the relaxation process, that has been done on *(i)* sub-queries, should be found as follows depending on the previous equations in (9), (10), (11), and (12):

$$\boldsymbol{cost_R} = \frac{\left(\sum_{i=1}^{n} \frac{(cost_{In})+(cost_{Ren})+(cost_{Del})+(cost_{Order})}{4}\right)}{\boldsymbol{i}} \quad (13)$$

Then the similarity is computed as follows:

$$\boldsymbol{sim(X, Q) = 1 - cost_R} \tag{14}$$

■

All the sub-queries are sorted according to their value of $\boldsymbol{sim(X, Q)}$, the higher similarity the sub query has, the higher order it takes.

Example (16): continue

To find the similarity of the query *Q1*, first find the value of $\boldsymbol{cost_R}$ as follows:

$$\boldsymbol{cost_R = \frac{\left(\left(\frac{\frac{1}{3}+0+0+\frac{1}{3}}{4}\right)+\left(\frac{\frac{1}{3}+\frac{1}{3}+0+0}{4}\right)+\left(\frac{\frac{1}{3}+0+0+0}{4}\right)\right)}{3}}$$

$$\boldsymbol{= 0.134}$$

And the similarity between the query *Q1* and the pre-specified XML document is:

$$\boldsymbol{sim(cdcatalog.xml, Q1) = 1 - 0.134 = 0.866}$$

4.6.5 Decompression

During all the previous stages no decompression required except when the query has to retrieve information about the data part of the document. In this case only the relevant containers were decompressed to answer the sub-query having that part of data.

To retrieve the relevant parts of the XML document to the user, all the retrieved, ranked containers were decompressed using the same decompression technique discussed in section 4.4 in this chapter.

Although there may be more than one relevant containers retrieved from one or more XML documents, all these containers were combined and decompressed into one XML document to perform one tree instead of a forest of multiple XML trees.

If the user needs more queries to be processed on the resulted document, then this document should be compressed first to be within the XML repository and then it can be used in its compressed version. This feature is called *composition* and is borrowed from XQuery, in which the retrieved information is stored in a temporary XML file for further retrieving.

4.7 Chapter Summary

This chapter sets forth the main features in the design of the *XCVQ* system which has the ability to compress and/or decompress an XML document without losing its data. The significant feature of *XCVQ* is its ability to retrieve information from the compressed version according to different kinds of queries and especially vague queries. This required an expansion of the existing XPath queries through adding certain features to provide it with the ability to answer imprecise queries.

CHAPTER 5 *XCVQ* Testing, Evaluation and Discussion

Since the testing and evaluation processes are part of SDM, this chapter illustrates the detailed testing of *XCVQ* and its ensuing evaluation. Because the *XCVQ* model consists of three main parts, *XCVQ-C, XCVQ-D, and XCVQ-QP*, the testing strategy will involve testing each stage on its own. This chapter describes the testing of the three parts of the *XCVQ* model.

5.1 Testing Strategy

For the purposes of testing the complete model, the testing strategies are to be specified first. The next sections describe the behaviour testing strategy used (state graph) and then the functional testing strategies (white and black boxes).

5.1.1 Testing XCVQ's *Behaviour*

For the purposes of testing the complete model, first the state diagram was defined to describe the behaviour of *XCVQ* and to implement the State Graph testing strategy (Beizer, 1990; Farrell-Vinay, 2008).

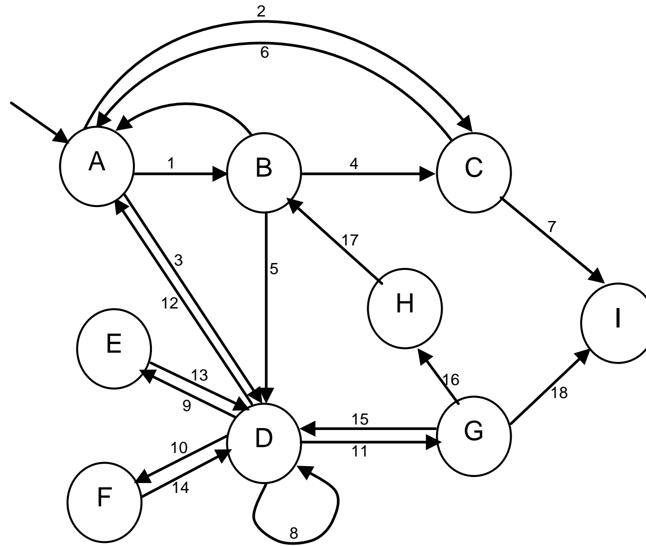


Figure 5-1: *XCVQ* State Graph

The following is the detailed description of each state of the state graph in Figure 5-1:

State-A: This state is the GUI of the designed model. It represents the starting state in order to deal with all the other states. This state has three outputs:

Out-1: to compress an XML document, go to state-B.

Out-2: to decompress an XML document, go to State-C. This output is true only if (Out-1) is performed at least one time.

Out-3: to write a query, go to State-D. This output is true only if (Out-1) is performed at least one time.

State-B: This state represents the process of compressing an XML document. It has two outputs:

Out-4: to decompress an XML document, go to State-C.

Out-5: to submit a query, go to State-D.

State-C: This stage represents the process of decompressing an XML document and it has two outputs:

Out-6: return to the starting state.

Out-7: submit the decompressed document to the user, go to State-I.

State-D: this is the most important state in the system which represents the query submission and checking its syntax. It has the following five outputs:

Out-8: if the submitted query has syntactical error(s), return to the same stage to resubmit another query.

Out-9: if the syntactically true query specifies the exact XML document to retrieve information from, go to Stage-E.

Out-10: if the syntactically true query does not specify the exact XML document to retrieve information from, go to Stage-F.

Out-11: take the out-of-errors query and the relevant XML document(s) as inputs to State-G.

Out-12: from this stage the user can return back to the starting state.

State-E: This state is responsible on retrieving the required XML document which specified by the query. It has only one output:

Out-13: carry the unique XML document which is specified by the query to State-D.

State-F: In the state, the set of relevant XML document is specified depending on the submitted query. This state has one output:

Out-14: carry the set of the relevant XML document(s) retrieved from the repository to State-D.

State-G: In this state, the query is processed and the required information is retrieved from the relevant XML document(s). It has three outputs:

Out-15: to ignore the current query, return to State-D.

Out-16: if more retrieval process required for the retrieved document, go to State-B to decompress the retrieved document first.

Out-18: to submit the results of the querying process to the user, go to State-I

State-H: This state returns the retrieved information as an XML document to the compressor to compress it and add it to the XML repository for further querying process and it has only one

Out-17: if the user required more querying on the retrieved information, go to State-B.

State-I: This state is the final state where the resulted document(s) are submitted to the user.

5.1.2 Testing *XCVQ*'s Structure & Functionality

Both White-Box and Black-Box testing strategies are used in order to test the structure and the functional of *XCVQ* respectively. In the White-Box testing strategy all the subroutines in the system were tested to check every single statement. Depending on this testing strategy, different kinds of XML documents were derived to guarantee that all paths, logical decisions, loops, and data structures have been tested at least once. Firstly, the complete *XCVQ* system was divided into three main sub-systems: *XCVQ-C, XCVQ-D, and XCVQ-QP* in order to make it easier to follow the white-box testing strategy. Secondly, each sub-system was divided into small units to follow the *unit white-box testing type*. For each unit three white-box tests were made:

(1) *Conditional test:* In this test all the condition statements were tested checking the values of the Boolean variables and the correctness of the conditions.

(2) Data lifecycle & data structure test: the second white-box tests the lifecycle of the variables, their initializations, their value changing, and their expiring. It also checks the created data structures by testing their boundaries, applicability, initializations, and updating their data.

(3) *Loop testing:* In this box all the loops in each unit were tested. The test includes the control variable initialization value, the truth of the control condition, the change in control variable, and the guarantee of its termination.

While the structure of the designed system is crucial to the White-Box testing strategy, it has no role in the Black-Box strategy since this strategy is

aimed at observing the outputs of the designed system for certain inputs. The main aim of this strategy is to test all the functional requirements, and hence it attempts to derive the necessary data for achieving that aim. In this chapter, the intensive test for the chosen XML data corpus and the independent test were both achieved.

During both previous strategies, a huge amount of XML data was used to cover different data ratios, depths, resources, and sizes. The overall tested data amounted to more than 1500 MB with 45 XML documents (see Appendix-C)

5.2 Testing Factors

To test the performance of the *XCVQ*, all the factors listed in Table 5-1 were used. The following is the complete description of these factors and their importance in the testing process:

Table 5-1: *XCVQ* Testing factors

Sub-system	**Testing factor**
XCVQ-C	- Structure Compression Ratio - Structure Compression Time - Compression Ratio - Compression Time
XCVQ-D	- Structure Decompression Time - Decompression Time
XCVQ-QP	- Functionality test - Performance Test

— *Compression Ratio (CR):* this factor is used to test the difference between the original XML file size and the compressed file size as illustrated in Eq.(15) (Salomon, 2007). It is used in two stages. In the first stage, only the structure part of the document was compressed and in the second stage the data and the structure parts were compressed. Depending on this factor, the relation between CR and the Data Ratio (DR) and the relation between CR and the size of the file were found. For this purpose a corpus of XML documents was used. Its complete description is discussed in the next paragraph.

$$CR = 1 - \left(\frac{Size\ of\ compressed\ file}{Size\ of\ original\ file}\right) \quad (15)$$

— *Compression Time (CT):* This factor is used to determine the time required to compress each XML document in seconds (s) and to specify its relation with the file size.

— *Decompression Time (DT):* This is the measure of the time required to decompress the XML document in order to obtain the original one. The effect of the file size on DT was obtained.

— *Query Functional Test (QFT):* The purpose of this test is to determine the main types of queries that can be processed by *XCVQ-QP*. For this purpose, a query benchmark was tested.

— *Query Performance Test (QPT):* This factor is used to determine the time required to process each of the XPath query in the benchmark and retrieve the relevant results.

All the time comparison factors shown in the following figures are scaled by ($\log_{10}$) to make the figures clearer. All the negative values in these figures mean that the actual time values were less than (1).

5.3 Data Preparation

To test the *XCVQ* model, a set comprising of different types of XML documents has been chosen. These documents should have different sizes, number of elements, number of nodes, the depth of the longest path, and the data ratio (DR) which is calculated as follows (Sakr, 2009):

$$\boldsymbol{DR_d = (D_d / Si_d)/100} \tag{16}$$

Where $\boldsymbol{DR_d}$ is the data ratio for the XML document (*d*), (*D*) is the data, and (*Si*) represents the size of the XML document.

According to their main characteristics, XML documents can be categorized into three types (Maneth et al., 2008; Sakr, 2009):

1. Textual documents (TD): The DR_d of this type of documents exceeds 70%. The structure of these documents is very simple. Books and articles are examples of this type.
2. Structural documents (SD): In this type of XML documents, the DR_d is less than 30%. Baseball box score and line-item shipping are two examples of this type.
3. Regular documents (RD): These documents have DR_d between 40% and 60%. Relational databases are examples of this type.

The complete descriptions of the XML corpus with all the required information and the detailed description of all the groups in the corpus are listed in Appendix-C.

5.4 Testing Environment

All the testing were carried out on a personal computer with Intel(R) Core(TM)2 Due CPU processor that has the speed of 5.50 GHz. The RAM memory of the tested environment is 4.00GB and 300GB of hard disk drive. It has 32-bit Windows Vista operating system.

5.5 XCVQ-C and XCVQ-D Testing

The testing technique for the *XCVQ-C* is made in two stages. The first stage is done by compressing only the structure of the XML document and creating the *path-dictionary* without compressing the data part of the document. The second stage is done by compressing the structure and the data parts to obtain the final XML compressed document which will be used in the querying process.

5.5.1 XCVQ-C and XCVQ-D Testing: Stage-1

The main purpose of this stage is to examine the effect of redundancy on the structure of the XML document and its overall size. In this stage, the data part of the document has not been compressed and thus keeps its original size, while the structure part is abridged and replaced with the elements index and the attribute name entries in the *path-dictionary*. The compressed XML document, at this stage, contains the *path-dictionary* and the created containers except that the data inside these containers are not compressed.

This test includes finding the Structure Compression Ratio (SCR), specifying the Structure Compression Time (SCT) and its relationship to the size

of the XML document, as well as determining the Structure Decompression Time (SDT) and its relationship to the size of the XML document.

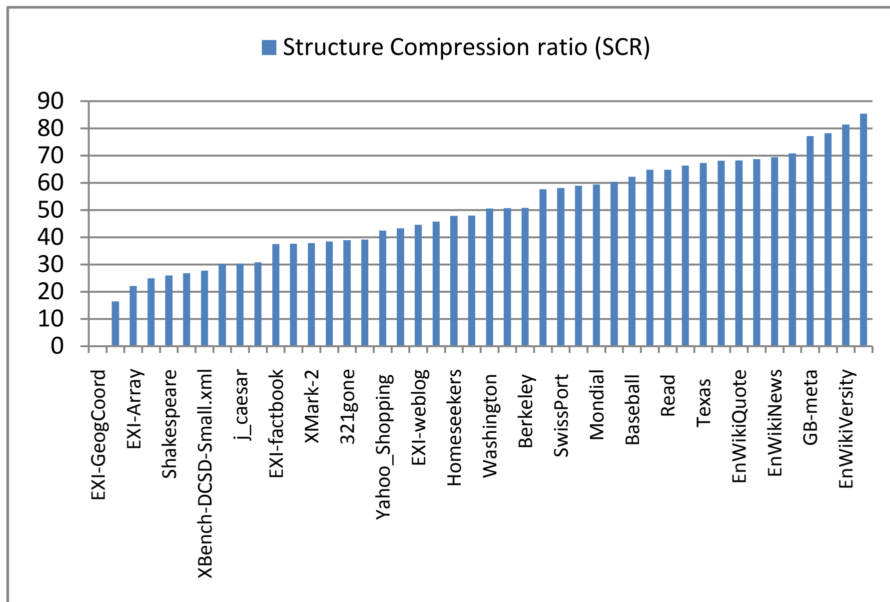


Figure 5-2: SCR for the XML corpus

Figure 5-2 explains the Structure Compression Ratio (SCR) for the XML corpus. By keeping the data in its original size and compressing only the structure part of each document, the resulted SCR is between *0.003* and *85.43* and the average SCR is *49.47*. The value of SCR depends on the structure ratio of each document, which is listed in appendix-D, and the repetition of the schema in this document. This test explains the role of the redundancy in the structure of the XML document.

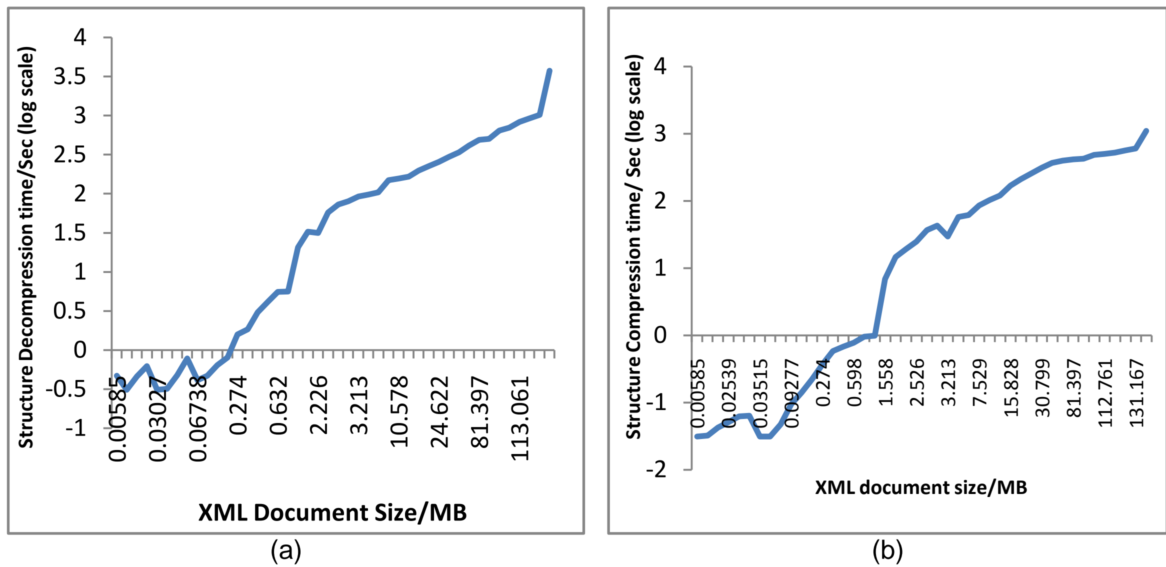


Figure 5-3: (a) Structure Compression Time for the XML corpus and (b) Structure Decompression Time for the XML corpus.

Figure 5-3-(a) shows the relation between the size of the XML document (X) and the structure compression time (Y), while Figure 5-3-(b) illustrates the time (Y) required for decompressing the XML document and restoring the original one. It is clear from the above figures that the relationships between the two variables in both cases are expanding almost linearly. The correlation coefficient between X and Y was $r = 0.886607$ in the compression case and $r = 0.996626$ in the decompression case. These values indicate the strong positive relationship between the size of the XML document in the one hand and the compression and decompression time on the other. The actual SCR, SCT, and SDT for the complete XML corpus are listed in Appendix-D.

5.5.2 XCVQ-C and XCVQ-D Testing: Stage-2

In this testing stage, the fully designed *XCVQ-C* and *XCVQ-D* were tested. The main aims of this test is to determine the average compression ratio for the XML corpus and the compression ratio for each of the documents, to specify the compression and decompression time and their relationship to the size of the XML document, and to generate the XML repository which is going to be used in the testing of *XCVQ-QP*.

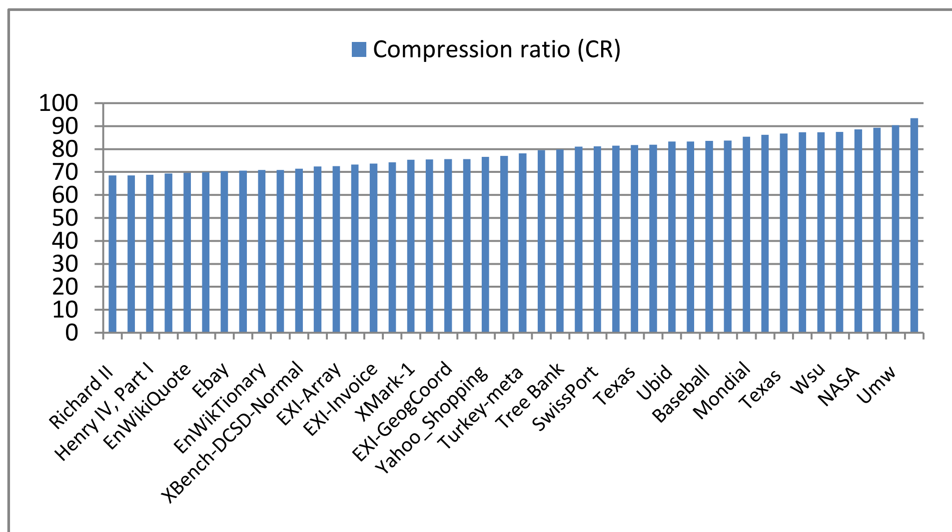


Figure 5-4: CR for the XML corpus

The compression ratio of the complete XML corpus is shown in Figure 5-4. The resulted compressed file contains the *path-dictionary* and the containers after compressing their data using Gzip back-end compressor. The minimum resulted compression ratio is *68.51* for Richard II and the maximum is *93.52* for Sweden-meta. The average compression ratio for the complete XML corpus is *78.45*.

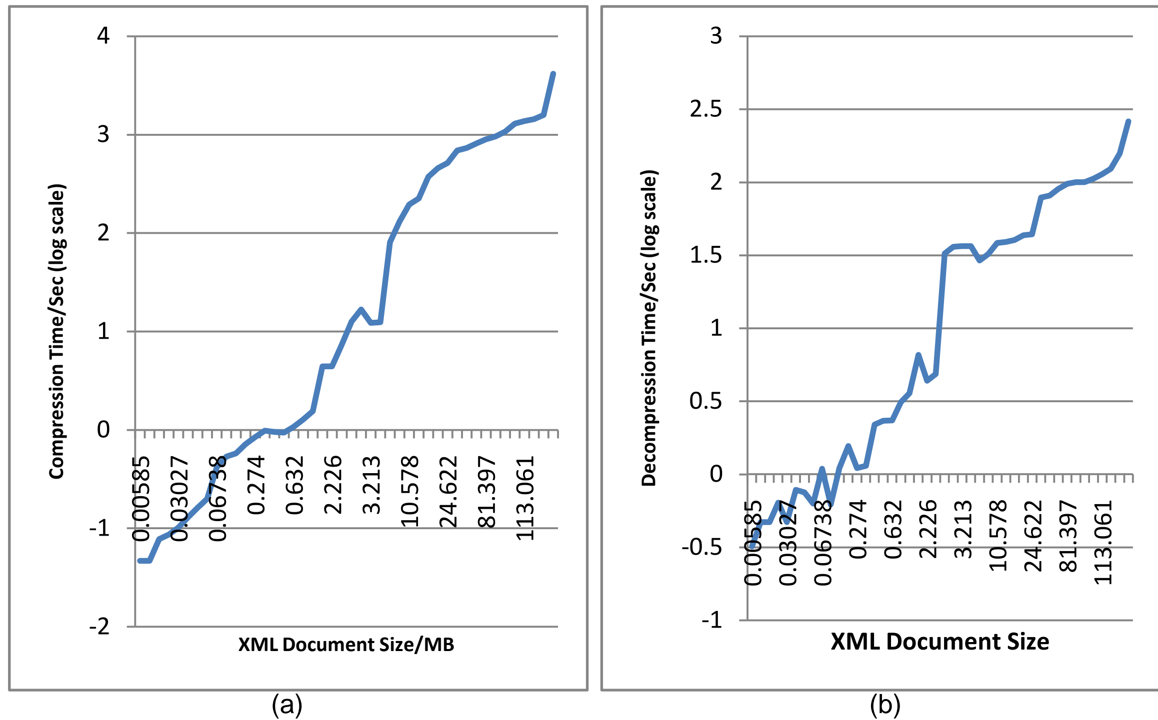


Figure 5-5: (a) Compression Time and (b) the Decompression Time for the XML corpus.

Figure 5-5-(a) shows the compression time for the complete XML corpus according to the size of the XML document while Figure 5-5-(b) shows the time required to decompress the XML documents. Again, the relationship between the compression/decompression time and the size of the XML document is almost linear and the correlation coefficient between the compression time and the XML document size is: r=0.971702, while it is r=0.888598 in the case of decompression. This illustrates the strong positive relation between the two tested variables. The complete tested files alongside with their CR, CT, and DT are listed in Appendix-D.

5.6 XCVQ-C & XCVQ-D Evaluation

For the purpose of evaluating *XCVQ-C* and *XCVQ-D*, comparisons were made between *XCVQ* and other competitive techniques. Depending on the availability of the techniques and the XML corpus used in the testing of these techniques, four queriable XML compressors were chosen for the purpose of

comparison: XGrind (Tolani and Haritsa, 2000), Xpress (Min et al., 2003), XQzip (Cheng and NG, 2004) , XQueC (Arion et al., 2007), and (Müldner et al., 2009). The XML corpus used in the testing and the compression ratio for each document is shown in Figure 5-6. The evaluation of the XCVQ includes comparing the following factors: CR, CT, and DT.

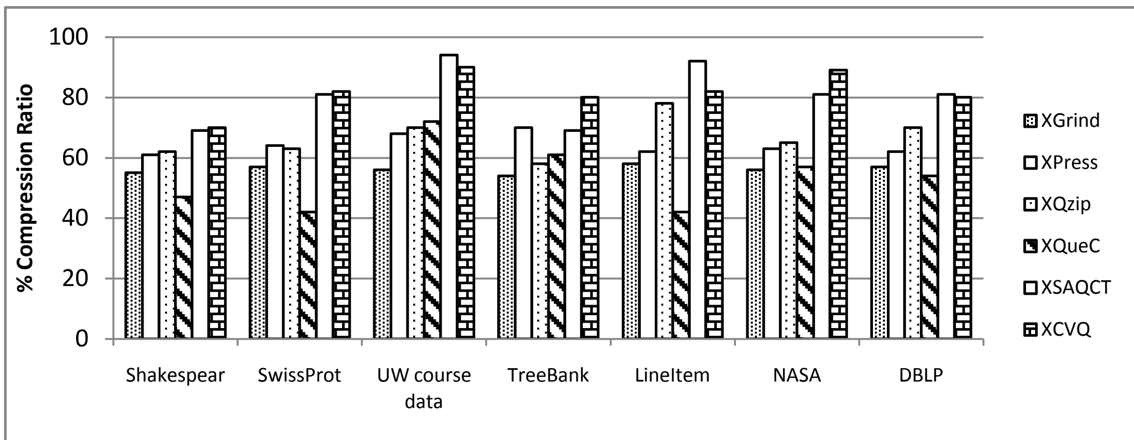


Figure 5-6: Evaluating *XCVQ-C* CR.

It is clear that *XCVQ-C* achieved a better compression ratio than other compressors except when dealing with high structural documents, since *XSAQCT* achieved better ratio. But when dealing with querying the compressed XML document, *XSAQCT* has the ability to answer only exact match queries since it transfers the structure of the document into an annotated-tree which can be compressed better than structured-tree. The average CR of the *XCVQ-C* is considered to be the best between all the other techniques for the selected documents, as listed in Table 5-2.

Table 5-2: Average CR for all the tested XML compressors.

XML compressor	Average CR
XGrind	57.39
XPress	57.55

XQzip	66.95
XQueC	68.4
XSAQCT	80.02
XCVQ	81.85

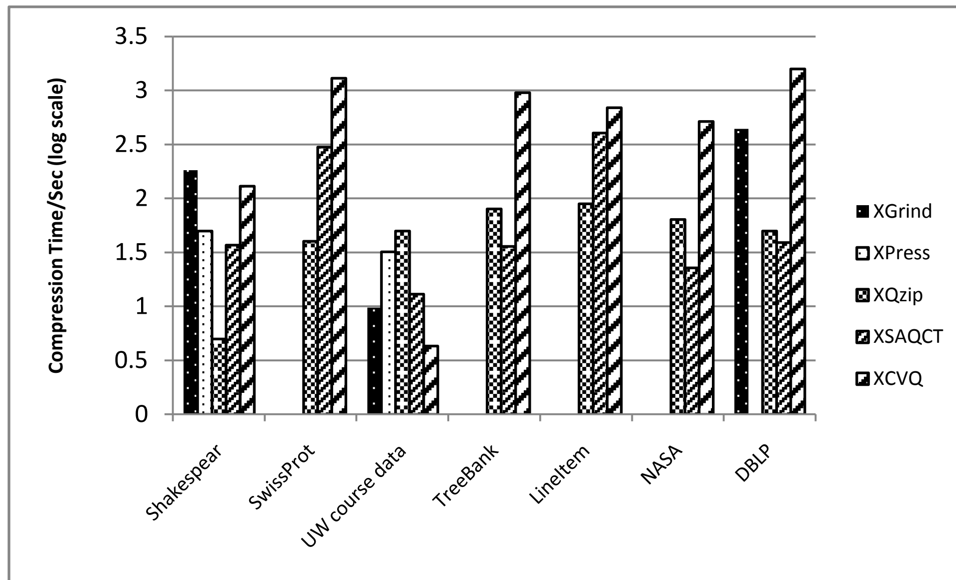


Figure 5-7: Evaluating *XCVQ-C* CT

As seen in Figure 5-7, the time required by the *XCVQ-C* to compress the XML document was higher than the other compressors in most cases. This is due to the SAX parser being used by *XCVQ-C*, which traverses the XML document only once, during which time the complete containers and the structured tree were constructed. While the time required to decompress and regenerate the XML document, shown in Figure 5-8, was better than some of the XML compressors.

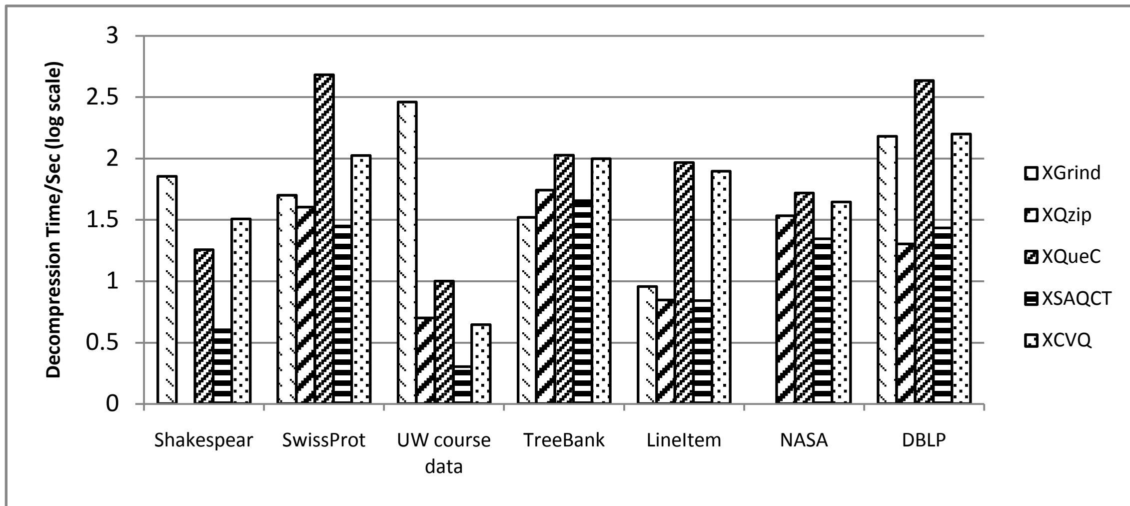


Figure 5-8: Evaluating *XCVQ-D* DT.

5.7 XCVQ-QP Testing

For the purpose of testing the performance of the *XCVQ-QP*, a XPath benchmark is used from XPathMark (Franceschet, 2005) since it covers all types of XML queries. The queries in this benchmark are divided into two main categories either Query Functional Test (*QFT*) or Query Performance Test (*QPT*).

5.7.1 QFT

XPath-FT queries are used to check the completeness and correctness of the query processor and are grouped into five aspects. Table 5-3 illustrates these five aspects. Since the main concern of *XCVQ-QP* is to process vague queries, only the vague cases in each of the aspects are tested. Since the third aspect could not be as vague, the testing process at this stage ignores this aspect. Furthermore,

one additional aspect was added to the existing aspects (Multi-File aspect) to test the ability of *XCVQ-QP* to retrieve the required information even if it is disseminated in more than one XML document.

Table 5-3: XPathMark-FT query benchmark

QFT concepts	Description
Axes	parent, descendant, preceding
Filters	predicates
Node Test	Comment(), text(), node()
Operators	Relational operators (<, =,…) and Boolean operators (and, or)
Functions	String manipulating functions and mathematical functions
Multi-File	Retrieving information from more than one XML document

Table C-2 in Appendix-C lists all the QFT concepts alongside with the queries associated with each concept by applying the example XML document in Figure 4-3 as a case study. All the listed queries were successfully processed by *XCVQ-QP* and retrieve the required information.

5.7.2 QPT

The QPT queries test the exact time required to answer a specific query (Franceschet, 2005). For this purpose, the same concepts in Table 5-4 were used to test the performance of the *XCVQ-QP* by testing the time required to process the set of queries for each concept and retrieve the information from a specific XML document chosen from the used XML corpus with different sizes.

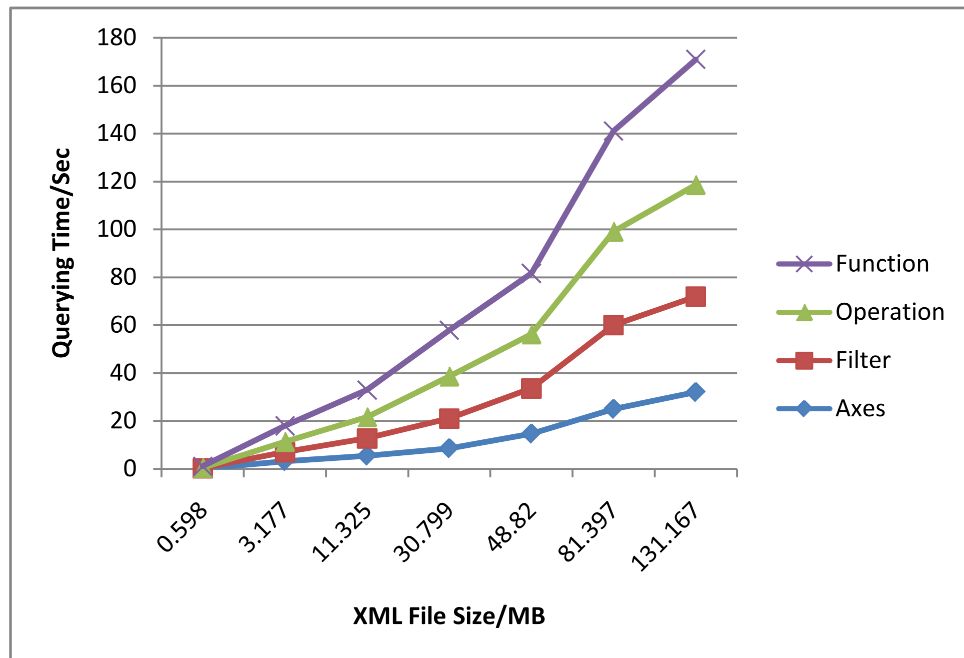


Figure 5-9: Testing *XCVQ-QP* Querying time

The testing results in Figure 5-9 includes all the concepts of the selected benchmark after averaging the time required to process the set of queries within each concept. These sets were applied to retrieve information from various XML documents with different sizes. It is clear from the aforementioned figure that the axes queries need less time to be processed than the other concepts. This is due to the structure of the compressed XML document which requires searching only the indexes of the containers to process these queries.

Since the queries belonging to the Filter concepts required partial decompression only for the retrieved containers, this set of queries needs more time than the queries in the first set. Because the set of queries in the Operation concept needs partial decompression for the relevant containers plus filtering the values in the retrieved information according to the given operation, they need even more time to be processed. Finally, the set of queries containing function calls require processing either the synonym or similarity of the given parameter which needs the highest time among other concepts as these functions require searching the dictionary or other similar data respectively.

Another test was made to check the performance of the queries in the last concept (multi-file). The test concludes that the time required to process a query from that set was dependent on several factors such as the size of the relevant documents, the number of relevant documents, and the size of the XML repository. It could thus be concluded that these entire factors have a positive relationship with the query processing time.

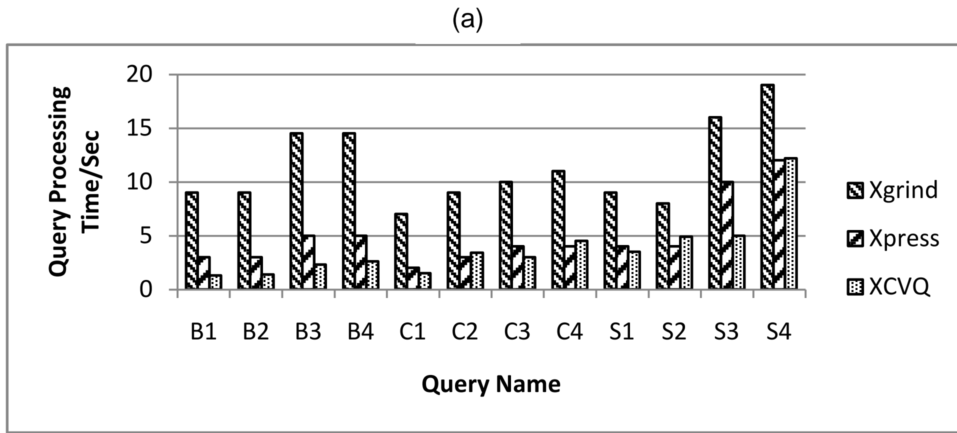
5.8 XCVQ-QP Evaluation

To evaluate *XCVQ-QP*, a test was first made to check the functionality of the model and its capability to process different kinds of queries. All the existing XML queriable compressors were tested to determine the types of queries each compressor can process. All the existing queriable compressors have the ability to process SQ, while some of them were designed to process specific types of queries. As discussed before, *XCVQ-QP* has the ability to process the vague queries plus all the other kinds of queries which renders it the only queriable XML compressor with such a feature.

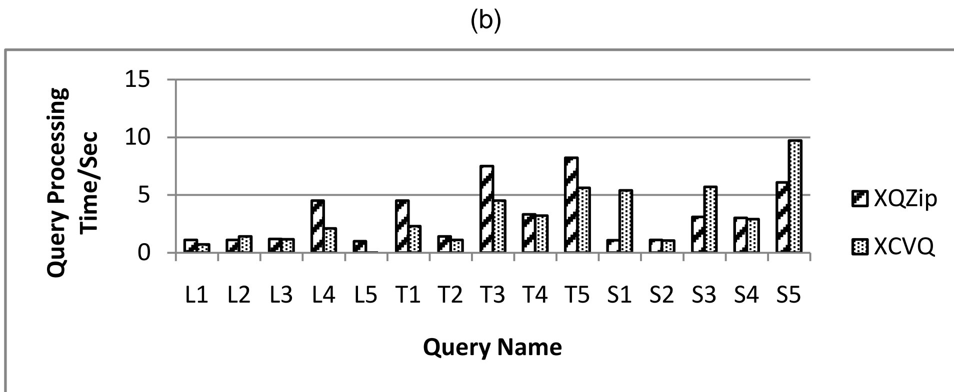
Another evaluation test was made to compare the time required to process a query and retrieve the relevant information accordingly. Since each of the previous XML compressors used a different set of queries and documents to test their querying time, several tests were made to compare *XCVQ-QP* with these compressors using their queries and XML document sets.

The evaluation tests were made to compare the querying time with XGrind and Xpress, XQZip, and XSAQCT using the set of queries and the XML documents listed in Appendix-E (Set-1) (Min et al., 2003), (Set-2) (Yang et al., 2006), and (Set-3) (Müldner et al., 2009) respectively.

(a)



(b)



(c)

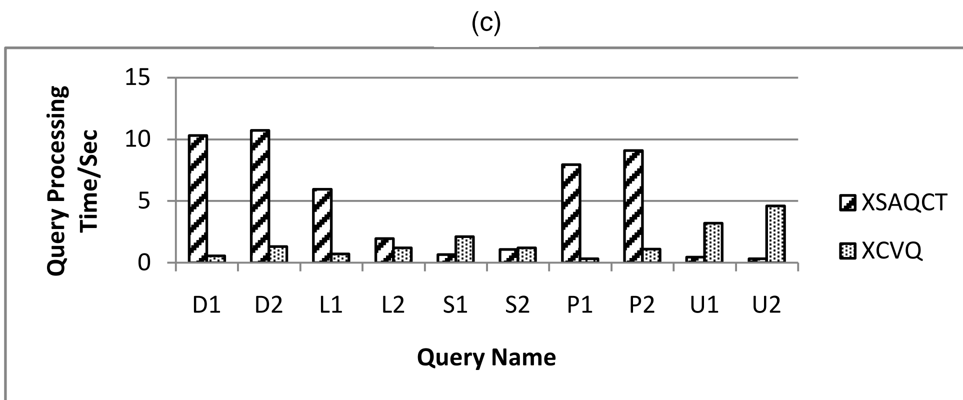


Figure 5-10: *XCVQ* Query processing time against (a) XGrind and Xpress, (b) XQZip, and (c) XSAQCT.

As declared in Figure 5-10, the time required to process the queries using *XCVQ-QP* was less than the time required to process the same queries using almost all the previous XML queriable compressors, except for the queries that require data retrieval such as the queries in the filter concept. This is due to the fact that *XCVQ-QP* needs to decompress the relevant containers in order to retrieve the required information. The time of *XCVQ-QP* was even more than the

other compressors when retrieving information from the textual XML documents since the size of the containers in these documents were higher than other types of documents.

The designed query processor is considered to be the first processor which has the ability to retrieve information according to all types of queries from the compressed XML documents. Comparing it with the techniques that retrieve information from the original XML documents, *XCVQ-QP* covers different sides of vague queries (path expansion, data value expansion, and function set expansion). On the other hand, the previous techniques are dedicated to solving only one side (path expansion as in (Grust, 2002; Amer-Yahia et al., 2004) and or two sides (path expansion and function expansion as in (Campi et al., 2009) or path expansion and data value expansion as in (Brisaboa et al., 2010).

5.9 Chapter Summary

In this chapter, extensive tests were carried out to check the performance and functional abilities of *XCVQ*. In the compressor part of the model, the model was tested using a corpus of XML documents that have different features. After comparing the compression ratio with other XML compressors, *XCVQ* showed better ratios in most of the tested documents and its average ratio was higher than all other tested techniques. On the other hand, the compression time was high and needs further development in the future. An independent test was also made to test the compression ratio of *XCVQ-C* and the results of the tested data are listed in Appendix-F.

From the decompression side, *XCVQ-D* was fast enough compared to the existing techniques and the decompressed documents were lossy when there were dummy elements in the XML document. The ratio of the dummy elements and that of the structure loss are listed in Appendix-G.

Finally, *XCVQ-QP* was tested to check for its ability to retrieve information according to several kinds of vague queries and other kinds of queries. A benchmark of queries was chosen and tested for the functional and

the performance abilities of the designed model. The results were very encouraging, since the model proved its ability to process different kinds of queries in a competitive processing time.

CHAPTER 6 Conclusions and future work

This thesis introduced a new model which has the ability to compress an XML document efficiently and retrieve information from the compressed file according to vague queries and even various other types of queries. This chapter will outline the main conclusions of the research as well as the main advantages and limitations of the designed model. Finally, the chapter will also list possible future trends in this research in terms of developing the proposed model.

6.1 Conclusion

As the importance of XML usage for storing and transferring data via the World Wide Web becomes increasingly clear, there is a corresponding need to compress the size of XML documents, dealing with them in their compressed mode so as to make them accessible to devices with limited resources. When these compressed documents are used by simple users, in a situation where there is absence of schema, or if such a user has no exact idea of what s/he is looking for, there should be a special technique available to adequately deal with these types of queries. The questions had been raised by this research and their answers are as follows:

1. *Is it possible to design a new compression technique that has the ability to compress the XML documents and achieve better compression ratio without the need to the document's Schema or its DTD?*

   The answer to this question is *XCVQ* compressor. The design of the model showed the best average compression ratio (78.45) among the other XML queriable compressors without the need to the XML schema to be available. This was due to several reasons, such as: (1) limiting the storage of each element and attribute name in the document to only one number, which represents the

order of that element or attribute in the XML document, instead of being two numbers, and (2) increase the granularity of the data to be compressed in order to perform better compression ratio. Although this design issue increased the compression ratio, but it affects the time required to compress the document by increasing this time to be higher than the time require to compress these documents using other techniques. However, the compression process usually made only once, while the querying process can be done hundreds of times to retrieve information from the compressed files.

2. *What is the influence of the structure redundancy on the overall size of the XML document?*

   To answer this question, *XCVQ Structure Compressor* was designed. In the compression process of the XML documents, the research found the strong affect of the redundancy in the structure of the document on its overall size. By succinctly storing the structure part of the XML document and keeping the data part as it is, the experiments showed good compression ratios which were up to 85.43 and averaged 49.47 for the tested XML corpus. This shows the big redundancy in the structure part of the document, apart which is considered to be very important for several purposes and retrieving information is one of them.

3. *What are the main types of vague queries and when they can be occur? Have the existing XPath query language the ability to answer vague queries? If no, what is the required expansion that should be made on XPath to give it this ability?*

   Vague queries are one of the important types of queries. They occur in different situations and require special ways to be processed since the existing query languages do not have the ability to answer these queries. The *XCVQ-QP* can deal with

simple and complex queries by forcing each query to pass by two decomposition stages in order to make it easier to retrieve information from the relevant document(s) and then combine the sub-results to be decompressed and submitted to the user. This process required the expansion of XPath query language in different sides: the path expansion, the data value expansion, and the set of functions expansion. The time required to process the queries are very competitive especially when dealing with structure-based queries, since the compressed structure of the document helps in accelerating the retrieving process.

4. *How to determine the relevant XML document(s) from thousands of documents without the need to scan them completely for time saving purposes? And is it possible to retrieve information from more than one XML document without the pre-specification of these documents using one XPath query?*

   Instead of scanning the complete document to search for a specific bit of data, *XCVQ-QP* uses the path-dictionary, which contains all the elements and attributes names, to specify the relevant documents from thousand of XML documents. In this way, it is now possible to retrieve information from unspecified document(s). While all the existing XML query processors required the user to pre-specify the required documents to retrieve information from them, *XCVQ-QP* has the ability to retrieve information from one or more than one XML document without the need to specify exactly which document could contain the required information.

6.2 Recommendations

— The main purpose of designing *XCVQ* is to process vague queries on compressed XML documents. For that reason, the first

recommendation for the model is to be used in cases where vague queries could be submitted, such as when dealing with naive users, where there is absence of schema, and when the required information is scattered among many files.

— The model is recommended to be used in retrieving information from XML documents when these documents have to be stored in devices with limited resources. The required documents can be compressed once and then queried several times with very limited resources requirements.

6.3 Future Work

Several research issues can be explored to improve the model:

— The model in this research can be developed to convert *XCVQ* into a complete XML management system with the ability to manage XML document in its compressed stage. The management process includes adding, deleting, or editing elements or attributes names. This process does not require any decompression, since the change is only made to the structure part of the document. The management process can include editing in this part of the document. In this case, only the container(s) with the required data should be decompressed using the Gzip back-end decompressor. They could also be used for editing the data and re-compressing the container(s).

— Another development is providing the ability to retrieve information from XML documents written in languages other than English. This could be done by adding a translator to translate any data part into other languages and retrieve the information accordingly.

— The model can be enriched by adding a Natural Language Processor that can convert a user's query into a vague XPath query and then

retrieve the required information from the compressed XML document.

- Remains to be fully implemented is the complete set of XPath statements such as “for” and “if”.

Publications

- Baydaa Al-Hamadani, Joan Lu. *Processing Vague Queries on Abridged XML Documents*. To be published in the journal of Philosophical Transaction of the Royal Society.
- Baydaa Al-Hamadani, Joan Lu, and Raad F. Alwan. *A new Schema-Independent XML Compression Technique.* Accepted for publication in the International Journal of Information Retrieval Research, 2011.
- Nael Hirzallah, Dead Al-Halabi, and Baydaa Al-Hamadani. *University Grades System Application using Dynamic Data Structure*. IJCSI Volume 8, Issue 1, January 2011.
- Daed Halabi, Nael Hirzallah, and Baydaa Al-Hamadani. *Dynamic Grading System for Universities.* 3rd International Conference on Advanced Computer Theory and Engineering, 2010.
- Baydaa T. Al-Hamadani, Raad F. Alwan, and Joan Lu, *XQPoint: A Queriable Homomorphic XML Compresso*r, IEEE ***6th International Conference on Innovations in Information Technology. Al-Ain, UAE, Page 100-104***. December PP: 15-17, 2009.
- Baydaa T. Al-Hamadani, Raad F. Alwan, Joan Lu, and Jim Yip, *Vague Content and Structure (VCAS) Retrieval for XML Electronic Healthcare Records (EHR)*, Proceeding of the 2009 International Conference on Internet Computing, USA, PP: 241-246, 2009.
- Baydaa T. Rashid, Raad F. Alwan, Joan Lu, and Jim Yip, *Recent Development in XML-IR*, proceeding of the School of Computing and Engineering Annual Researchers' Conference, University of Huddersfield, UK, PP: 106-109, 2008.

Reference List

GZip Compressor, . http://www.gzip.org/.

Yousof, M. M., Shukur, Z. & Abdullah, A. L. (2011) CuQuP: A Hybrid Approach for Selecting Suitable Information Systems Development Methodology *Information Technology Journal*.

Al-Hamadani, B., Lu, J. & Alwan., R. F. (2011) A new Schema-Independent XML Compression Technique. *Accepted for publication in the International Journal of Information Retrieval Research*.

Al-Hamadani, B. T., Alwan, R. F., Lu, J. & Yip, J. 2009. Vague Content and Structure (VCAS) Retrieval for XML Electronic Healthcare Records (EHR). Proceeding of the 2009 International Conference on Internet Computing, USA. P: 241-246.

Al-Khalif a, S., Jagadish, H., Patel, J., Wu, Y., Koudas, N. & Srivastava, D. (2002). Structural Joins: A Primitive for Efficient XML Query Pattern Matching. 8th International Conference on Data Engineering, San Jose, CA, USA.

Alistair, M., Radford, M. N. & Ian, H. W. (1998) Arithmetic coding revisited. *ACM Trans. Inf. Syst.,* 16**,** 256-294.

Amer-Yahia, S., Lakshmanan, L. V. S. & Pandit, S. 2004. FleXPath: Flexible Structure and FullText Querying for XML. ACM, SIGMOD., Paris, France. PP: 83-94.

Amir-Yahya, S., Cho, S. & Srivatava, D. (2002). Tree Pattern Relaxation. EDBT 8th International Conference on Extending Database Technology, Prague, Czech Republic.

Anders, M., (2009). *An Introduction to XML and Web Technologies*, Pearson Education.

Arion, A., Bonifati, A., Manolescu, I. & Pugliese, A. (2007) XQueC: A query-conscious compressed XML database. *ACM Trans. Internet Technol.,* Vol. 7**,** 10.

Arroyuelo, D., Claude, F., Maneth, S., M¨Akinen, V., Navarro, G., Nguyen, K., Sir´En, J. & V¨Alim¨Aki, N. 2010. Fast In-Memory XPath Search using Compressed Indexes. In Proceedings of the IEEE Twenty-Sixth International Conference on Data Engineering (ICDE 2010), California, USA.

Augeri, C. (2008), *On Some Results in Unmanned Aerial Vehicle Swarms*, Ph.D Thesis, Air Force Institute of Technology, San Diego, CA, USA.

Augeri, C. J., Bulutoglu, D. A., Mullins, B. E., Baldwin, R. O. & Leemon C. Baird, I. (2007). An analysis of XML compression efficiency. Proceedings of the 2007 workshop on Experimental computer science, ACM, San Diego, California. 7.

Beizer, B., (1990). *Software Testing Techniques, Second Edition*.

Bodenhofer, U. & Küng, J. 2001. Enriching vague queries by fuzzy orderings. European Society for Fuzzy Logic and Technology - EUSFLAT, Pages 360-364.

Bonifati, A., Lorusso, M. & Sileo, D. (2009) XML lossy text compression: A preliminary study. *Lecture Notes in Computer Science*, Pages: 106-113.

Bourret, R. (2005) XML and Databases. Ronald Bourret.

Brisaboa, N. R., Cerdeira-Pena, A., Navarro, G. & Pasi, G. 2010. An Efficient Implementation of a Flexible XPath Extension. Recherche d'Information Assistee par Ordinateur - RIAO, Pages 140-147.

Buneman, P., Grohe, M. & Koch, C. (2003) Path Queries on Compressed XML. IN JOHANN-CHRISTOPH, F., PETER, L., SERGE, A., MICHAEL, C., PATRICIA, S. & ANDREAS, H. (Eds.) *Proceedings 2003 VLDB Conference.* San Francisco, Morgan Kaufmann.

Bzip2 (1996) http://www.bzip.org/.

Campi, A., Damiani, E., Guinea, S., Marrara, S., Pasi, G. & Spoletini, P. (2009) A fuzzy extension of the XPath query language. *Journal of Intelligent Information Systems,* 33, 285-305.

Cheney, J. (2001). Compressing XML with multiplexed hierarchical models. IEEE Data Compression Conference (DCC 2001), IEEE Computer Society, pages 163-172.

Cheney, J. (2005). An Empirical Evaluation of Simple DTD Conscious Compression Techniques. Eighth International Workshop on the Web and Databases (WebDB 2005), Maryland, USA.

Cheng, J. & Ng, W. 2004. XQZip: Querying Compressed XML using Structural Indexing. International Conference on Extending Data Base Technology (EDBT),

Cleary, J. & Witten, I. (1984) Data Compression Using Adaptive Coding and Partial String Matching. *Communications, IEEE Transactions,* 32, 396 - 402.

Damiani, E. & Tanca, L. 2000. Blind Queries to XML Data. Proceedings of the 11th international conference on database and expert systems applications, . pp: 345-356.

De Meo, P., Palopoli, L., Quattrone, G. & Ursino, D. (2007) Combining Description Logics with synopses for inferring complex knowledge patterns from XML sources. *Information Systems,* 32**,** 1184-1224.

Dutta, A. K., Idwan, S. & Biswas, R. (2009) A Study of Vague Search to Answer Imprecise Query. *International Journal of Computational Cognition(IJCC),* Volume 7 Pages 70-75.

European, E., Agancy, (2003) http://www.eea.europa.eu/data-and-maps/data/airbase-the-european-air-quality-database-1

Exi (2009) http://www.w3.org/XML/EXI/.

Farrell-Vinay, P., (2008). *Manage Software Testing*, Auerbach.

Fazzinga, B., Flesca, S. & Pugliese, A. (2009) Retrieving XML Data from Heterogeneous Sources through Vague Querying. *ACM Transactions on Internet Technology,* Vol. 9**,** pages: 7-35.

Ferragina, P., Luccio, F., Manzini, G. & Muthukrishnan, S. (2006). Compressing and searching XML data via two zips. Proceedings of the 15th international conference on World Wide Web, ACM, Edinburgh, Scotland. 751-760.

Florescu, D., Kossmann, D. & Manolescu, I. (2000) Integrating keyword search into XML query processing. *Computer Networks,* 33**,** 119-135.

Franceschet, M. (2005) XPathMark: Functional and Performance Tests for XPath. *Lecture Notes in Computer Science, Springer,* vol. 3671**,** 129--143.

Fredrick, E. J. T. & Dr.G.Radhamani (2009) Fuzzy Logic Based XQuery operations for Native XML Database Systems. *International Journal of Database Theory and Application,* Vol. 2.

Fuhr, N. 1999. A probabilistic framework for vague queries and imprecise information in databases. 16TH INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES

Fuhr, N., Lalmas, M. & Trotman, A., (2006). *Comparative evaluation of XML information retrieval systems, 5th Edition*, Springer.

Gerlicher, A. R. S. (2007), *Developing Collaborative XML Editing Systems*, PhD thesis, University of the Arts London, London.

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

Goldberg, K. H., (2009). *XML: Visual QuickStart Guide Second* Peachpit Press-Pearson Education.

Groppe, J. (2008), *SPEEDING UP XML QUERYING*, PhD thesis, Zugl Lübeck University, Berlin.

Grust, T. (2002). Accelerating XPath location steps. ACM SIGMOD International Conference on Management of Data, Madison, WI, USA.

Gzip (1992) http://www.gzip.org/.

Halverson, A., Burger, J., Galanis, L., Kini, A., Krishnamurthy, R., Rao, A., Tian, F., Viglas, S., Wang, Y., Naughton, J. & Dewitt, D. (2003). Mixed Mode XML Query Processing. 29th International Conference on Very Large Data Bases, Berlin, Germany.

Harrusi, S., Averbuch, A. & Yehudai, A. 2006. XML Syntax Conscious Compression. Proceedings of the Data Compression Conference (DCC'06),

Hevner, A., March, S., Park, J. & Ram, S. (2004) Design Science in Information Systems Research. *MIS Quarterly,* Volume 28**,** pages 75-105.

Holman, G. K., (2002). *XSLT and XPath*, Prentice Hall PTR.

Huh, S. Y., Moon, K. H. & Lee, H. (2000) A data abstraction approach for query relaxation. *Information and Software Technology,* 42**,** 407-418.

Hung, P. C. K., (2009). *Services and Business Computing Solution with XML*, IGI Global.

Hunter, D., (2000). *Beginning XML*, Wrox Press Ltd.

Jiaheng, L. (2006), *Efficient Processing of XML TWIG Pattern Matching*, PhD thesis, NATIONAL UNIVERSITY OF SINGAPORE, SINGAPORE.

Kay, M., (2004). *XPath 2.0 Programmers Reference*, Wiley Publishing, Inc.

Kay, M., (2008). *XSLT 2.0 and XPath 2.0 Programmer's Reference*, Wiley Publishing, Inc.

Lalmas, M. & ROlleke, T. 2004. Modelling Vague Content and Structure Querying in XML Retrieval with a Probabilistic Object-Relational Framework Proceedings of the 6th International Conference on Flexible Query Answering Systems, FQAS volume 3055 of Lecture Notes in Computer Science, Springer. Pages 432-445.

League, C. & Eng, K. (2007). Schema-Based Compression of XML Data with Relax NG. IEEE data compression conference (DCC), Utah.

Li, H.-G., Aghili, S. A., Agrawal, D. & Abbadi, A. E. 2006. FLUX: Fuzzy Content and Structure Matching of XML Range Queries. In Proceeding of WWW 2006 Edinburgh, Scotland.

Liefke, H. & Suciu, D. 2000. XMill: an Efficient Compressor for XML Data. ACM,

Mandreoli, F., Martoglia, R. & Tiberio, P. 2004. Approximate Query Answering for a Heterogeneous XML Document Base. Proceedings of the 5th int. conf on web information systems engineering. , Brisbane, Australia.

Maneth, S., Mihaylov, N. & Saker, S. 2008. XML Tree Structure Compression. XANTEC'08, IEEE Computer Society,

Manning, C. D., Raghavan, P. & Schütze, H., (2008). *Introduction to Information Retrieval*, Cambridge University Press.

Mcgovern, J., Bothner, P., Cagle, K., Nagarajan, V. & Linn, J., (2003). *XQuery: Kick Start*, SAMS Publishing.

Mclaughlin, B. & Edelson, J., (2006). *Java and XML Third Edition*, O'Reilly.

Meersman, R., Tari, Z., Herrero, P., Abdelaziz, T., Elammari, M. & Branki, C. (2008) MASD: Towards a Comprehensive Multi-agent System Development Methodology. *On the Move to Meaningful Internet Systems: OTM 2008 Workshops.* Springer Berlin / Heidelberg.

Mihajlovic, V., Hiemstra, D. & Blok, H. E. (2006) Vague Element Selection and Query Rewriting for XML Retrieval. *Proceedings of the 6th Dutch-Belgian Information Retrieval Workshop (DIR 2006).* Delft, The Netherlands.

Min, J.-K., Park, M.-J. & Chung, C.-W. (2003). XPRESS: a queriable compression for XML data. Proceedings of the 2003 ACM SIGMOD international conference on Management of data, ACM, San Diego, California. 122-133.

Min, J.-K., Park, M.-J. & Chung, C.-W. (2009) Method of Performing Queriable XML Compression using Reverse Arithmetic Encoding and Type Inference Engine. IN PATENT, U. S. (Ed. USA, Korea Advanced Institute of Science and Technology.

Moffat., A. (1990) Implementing the PPM data compression scheme. *IEEE Trans. on Comm.,* 38(11)**,** 1917–1921.

Moro, M. M., Ale, P., Vagena, Z. & Tsotras, V. J. 2008. XML Structural Summaries. PVLDB '08, Auckland, New Zealand.

Morrison, J. & George, J. (1995) Exploring the Software Engineering Component in MIS Research. *Communication of the ACM,* Volume 38**,** pages 80-91.

Müldner, T., Fry, C., Miziołek, J. K. & Durno, S. 2009. XSAQCT: XML Queryable Compressor. Balisage: The Markup Conference 2009,

Murray, J. D. & Vanryper, W., (1996). *Encyclopedia of graphics file formats*, O'Reilly.

Ng, W., Lam, W.-Y. & Cheng, J. (2006) Comparative Analysis of XML Compression Technologies. *World Wide Web: Internet and Web Information Systems,* Vol. 9**,** Pages 5-33.

Norbert, F. & Kai, G. (2004) XIRQL: An XML query language based on information retrieval concepts. *ACM Trans. Inf. Syst.,* 22**,** 313-356.

Nunamaker, J., Chen, M. & Purdin (1991) Systems Development in Information Systems Research. *Journal of Management Information Systems,* Volume 7**,** pages 89-106.

Paparizos, S., Al-Khalifa, S., Chapman, A., Jagadish, H., V., Lakshmanan, L. V. S., Nierman, A., Patel, J. M., Sirvastava, D., Wiwatwattana, N., Wu, Y. & Yu, C. (2003). TIMBER: A Native System for Querying XML. ACM SIGMOD International Conference on Management of Data, ACM, San Diego, CA, USA.

Pkware (2003) http://www.pkware.com.

Plays, S. (2000) http://www.cafeconleche.org/examples/shakespeare/.

Rajpal, S., Doja, M. N. & Biswas, R. (2007) A method of vague search to answer queries in relational databases. *Information-an International Interdisciplinary Journal,* 10**,** 865-880.

Ray, E. T., (2001). *Learning XML Guide to Creating Self-Describing Data*, O'Reilly Media Inc.

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

Salomon, D., (2007). *Data Compression: The Complete Reference*, Springer.

Sanz, I. (2007), *Flexible Technique for Heterogeneous XML Data Retrieval*, PhD Thesis, Universitat Jaume, Spain.

Schlieder, T. 2001. Similarity Search in XML Data using Cost-Based Query Transformations. In Proceeding of ACM SIGMOD WebDB, pp. 19-24.

Schmidt, A., Waas, F., Kersten, M. & Carey, M. J. 2002. XMark: A Benchmark for XML Data Management. Proceedings of the 28th VLDB Conference, Hong Kong, China.

Shannon, C. E. (1948) A mathematica theory of communication. *Bell System Technical Journal,* 27**,** Pages: 379-423.

Sigurbjornsson, B. & Trotman, A. 2003. Queries: INEX 2003 working group report. 2nd workshop of the initiative for the evaluation of XML retrieval (INEX),

Skibinski, P., Grabowski, S. & Swacha, J. (2007). Effective Asymmetric XML Compression. CADSM,

Stamatina, B., Mounia, L., Anastasios, T. & Theodora, T. (2006). User expectations from XML element retrieval. Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, Seattle, Washington, USA.

Stasiu, R. K., Heuser, C. A. & Da Silva, R. (2005) Estimating recall and precision for vague queries in databases. IN PASTOR, O. & CHUNHA, J. F. E. (Eds.) *Advanced Information Systems Engineering, Proceedings.* Berlin, Springer-Verlag Berlin.

Tidwell, D., (2008). *XSLT*, O'Reilly.

Tolani, P. M. & Haritsa, J. R. (2000). XGRIND: A Query-friendly XML Compressor. IEEE 18th international conference on Data Engineering,

Trotman, A. & Sigurbjornsson, B. 2005. Narrowed Extended XPath I (NEXI) Advances in XML Information Retrieval Berlin / Heidelberg. Pages 16-40.

Violleau, T. (2001) Java Technology and XML. ORACLE.

W3c (1999) XML Path Language (XPath) - Version 1.0. W3C.

W3c (2001) XML Linking Language (XLink) Version 1.0. W3C.

W3c (2002) XML Pointer Language (XPointer). W3C.

W3c (2007a) XML Path Language (XPath) 2.0. World Wide Web Consortium.

W3c (2007b) XQuery 1.0 and XPath 2.0 Functions and Operators. W3C.

W3c (2008) Extensible Markup Language (XML) 1.0 (Fifth Edition).

W3c (2010a) XML Path Language (XPath) 2.0. World Wide Web Consortium.

W3c (2010b) XQuery 1.0 and XPath 2.0 Functions and Operators. W3C.

W3schools.Com (2006a) XLink and XPointer.

W3schools.Com (2006b) XML Examples.

Washington (2002) http://www.cs.washington.edu/research/xmldatasets/data/.

Waterloo (2003) http://softbase.uwaterloo.ca/~ddbms/projects/xbench/index.html.

White, S. (2008) How to Strike a Match. Catalys Web Site.

Wikipedia (2001) http://download.wikipedia.org/enwikinews/.

Williams, I., (2009). *Beginning XSLT and XPath: Transforming XML Documents and Data*, Wrox Press.

Wintertree (2006) http://www.wintertree-software.com/index.html. Canada.

Winzip (1990) http://www.winzip.com/.

Yang, Y., Ng, W., Lau, H. L. & Cheng, J. 2006. An Efficient Approach to Support Querying Secure Outsourced XML Information. In Proceedings of the 18th Conference on Advanced Information Systems Engineering (CAiSE), pages 157-171.

Zadeh, L. A. (1965) Fuzzy sets. *Information and control,* Vol. 8**,** pp: 338-353.

Zhang, Q. & Kankanhalli, M. S. 2003. Semantic video annotation and vague query. 9th International Conference on Multimedia Modelling (MMM 2003)

Zhao, F. & Ma, Z. M. 2009. Vague Query Based on Vague Relational Model. AISC 229-238.

Appendix-A: XPath's EBNF

The complete EBNF of the XPath query language is listed in this appendix. This form had been used in *XCVQ* to check the correctness of the syntax of the submitted query.

```
[1]   XPath                 ::=  Expr
[2]   Expr                  ::=  ExprSingle ("," ExprSingle)*
[3]   ExprSingle            ::=  ForExpr
                                 | QuantifiedExpr
                                 | IfExpr
                                 | OrExpr
[4]   ForExpr               ::=  SimpleForClause "return" ExprSingle
[5]   SimpleForClause       ::=  "for" "$" VarName "in" ExprSingle ("," "$" VarName "in"
                                 ExprSingle)*
[6]   QuantifiedExpr        ::=  ("some" | "every") "$" VarName "in" ExprSingle ("," "$"
                                 VarName "in" ExprSingle)* "satisfies" ExprSingle
[7]   IfExpr                ::=  "if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
[8]   OrExpr                ::=  AndExpr ( "or" AndExpr )*
[9]   AndExpr               ::=  ComparisonExpr ( "and" ComparisonExpr )*
[10]  ComparisonExpr        ::=  RangeExpr ( (ValueComp
                                 | GeneralComp
                                 | NodeComp) RangeExpr )?
[11]  RangeExpr             ::=  AdditiveExpr ( "to" AdditiveExpr )?
[12]  AdditiveExpr          ::=  MultiplicativeExpr ( ("+" | "-") MultiplicativeExpr )*
[13]  MultiplicativeExpr    ::=  UnionExpr ( ("*" | "div" | "idiv" | "mod") UnionExpr )*
[14]  UnionExpr             ::=  IntersectExceptExpr ( ("union" | "|") IntersectExceptExpr )*
[15]  IntersectExceptExpr   ::=  InstanceofExpr ( ("intersect" | "except") InstanceofExpr )*
[16]  InstanceofExpr        ::=  TreatExpr ( "instance" "of" SequenceType )?
[17]  TreatExpr             ::=  CastableExpr ( "treat" "as" SequenceType )?
[18]  CastableExpr          ::=  CastExpr ( "castable" "as" SingleType )?
[19]  CastExpr              ::=  UnaryExpr ( "cast" "as" SingleType )?
[20]  UnaryExpr             ::=  ("-" | "+")* ValueExpr
[21]  ValueExpr             ::=  PathExpr
[22]  GeneralComp           ::=  "=" | "!=" | "<" | "<=" | ">" | ">="
[23]  ValueComp             ::=  "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[24]  NodeComp              ::=  "is" | "<<" | ">>"
[25]  PathExpr              ::=  ("/" RelativePathExpr?)
                                 | ("//" RelativePathExpr)
                                 | RelativePathExpr
[26]  RelativePathExpr      ::=  StepExpr (("/" | "//") StepExpr)*
[27]  StepExpr              ::=  FilterExpr | AxisStep
[28]  AxisStep              ::=  (ReverseStep | ForwardStep) PredicateList
[29]  ForwardStep           ::=  (ForwardAxis NodeTest) | AbbrevForwardStep
```

```
[30]  ForwardAxis            ::=  ("child" "::")
                                  | ("descendant" "::")
                                  | ("attribute" "::")
                                  | ("self" "::")
                                  | ("descendant-or-self" "::")
                                  | ("following-sibling" "::")
                                  | ("following" "::")
                                  | ("namespace" "::")
[31]  AbbrevForwardStep      ::=  "@"? NodeTest
[32]  ReverseStep            ::=  (ReverseAxis NodeTest) | AbbrevReverseStep
[33]  ReverseAxis            ::=  ("parent" "::")
                                  | ("ancestor" "::")
                                  | ("preceding-sibling" "::")
                                  | ("preceding" "::")
                                  | ("ancestor-or-self" "::")
[34]  AbbrevReverseStep      ::=  ".."
[35]  NodeTest               ::=  KindTest | NameTest
[36]  NameTest               ::=  QName | Wildcard
[37]  Wildcard               ::=  "*"
                                  | (NCName ":" "*")
                                  | ("*" ":" NCName)
[38]  FilterExpr             ::=  PrimaryExpr PredicateList
[39]  PredicateList          ::=  Predicate*
[40]  Predicate              ::=  "[" Expr "]"
[41]  PrimaryExpr            ::=  Literal | VarRef | ParenthesizedExpr | ContextItemExpr |
                                  FunctionCall | FunctionCallList
                                  FunctionCallList ::= "synonyms" " ("  StrinLiteral ")"
                                  | "similar("  StrinLiteral ")"
                                  | "avg("  pathExpr ")"
                                  | "median" " (" pathExpr ")"
                                  | "between" " (" IntegerLiteral
                                          | DecimalLiteral
                                          | DoubleLiteral "," IntegerLiteral
                                          | DecimalLiteral
                                          | DoubleLiteral ")"

[42]  Literal                ::=  NumericLiteral | StringLiteral
[43]  NumericLiteral         ::=  IntegerLiteral | DecimalLiteral | DoubleLiteral
[44]  VarRef                 ::=  "$" VarName
[45]  VarName                ::=  QName
[46]  ParenthesizedExpr      ::=  "(" Expr? ")"
[47]  ContextItemExpr        ::=  "."
[48]  FunctionCall           ::=  QName "(" (ExprSingle ("," ExprSingle)*)? ")"
[49]  SingleType             ::=  AtomicType "?"?
[50]  SequenceType           ::=  ("empty-sequence" "(" ")")
                                  | (ItemType OccurrenceIndicator?)
[51]  OccurrenceIndicator    ::=  "?" | "*" | "+"
[52]  ItemType               ::=  KindTest | ("item" "(" ")") | AtomicType
[53]  AtomicType             ::=  QName
[54]  KindTest               ::=  DocumentTest
```

```
                                       | ElementTest
                                       | AttributeTest
                                       | SchemaElementTest
                                       | SchemaAttributeTest
                                       | PITest
                                       | CommentTest
                                       | TextTest
                                       | AnyKindTest
[55]  AnyKindTest                ::=   "node" "(" ")"
[56]  DocumentTest               ::=   "document-node" "(" (ElementTest | SchemaElementTest)? ")"
[57]  TextTest                   ::=   "text" "(" ")"
[58]  CommentTest                ::=   "comment" "(" ")"
[59]  PITest                     ::=   "processing-instruction" "(" (NCName | StringLiteral)? ")"
[60]  AttributeTest              ::=   "attribute" "(" (AttribNameOrWildcard ("," TypeName)?)? ")"
[61]  AttribNameOrWildcard       ::=   AttributeName | "*"
[62]  SchemaAttributeTest        ::=   "schema-attribute" "(" AttributeDeclaration ")"
[63]  AttributeDeclaration       ::=   AttributeName
[64]  ElementTest                ::=   "element" "(" (ElementNameOrWildcard ("," TypeName
                                       "?"?)?)? ")"
[65]  ElementNameOrWildcard      ::=   ElementName | "*"
[66]  SchemaElementTest          ::=   "schema-element" "(" ElementDeclaration ")"
[67]  ElementDeclaration         ::=   ElementName
[68]  AttributeName              ::=   QName
[69]  ElementName                ::=   QName
[70]  TypeName                   ::=   QName
[71]  IntegerLiteral        ::=   Digits
[72]  DecimalLiteral        ::=   ("." Digits) | (Digits "." [0-9]*)
[73]  DoubleLiteral         ::=   (("." Digits) | (Digits ("." [0-9]*)?)) [eE] [+-]? Digits
[74]  StringLiteral         ::=   ('"' (EscapeQuot | [^"])* '"') | ("'" (EscapeApos | [^'])* "'")
[75]  EscapeQuot            ::=   '""'
[76]  EscapeApos            ::=   "''"
[77]  Comment               ::=   "(:" (CommentContents | Comment)* ":)"
[78]  QName                 ::=   [http://www.w3.org/TR/REC-xml-names/#NT-QName]^Names
[79]  NCName                ::=   [http://www.w3.org/TR/REC-xml-names/#NT-NCName]^Names
[80]  Char                  ::=   [http://www.w3.org/TR/REC-xml#NT-Char]^XML
[81]  Digits                ::=   [0-9]+
[82]  CommentContents       ::=   (Char+ - (Char* ('(:' | ':)') Char*))
```

Appendix-B: Implementing XCVQ

The implementation of *XCVQ* is done using *Eclipse* environment for java programming language. The GUI for the system uses the (visualswing4eclipse) plug-in to makes the design more powerful, friendly, and easy to use. The main window of the system is shown in Figure B-1. Using this GUI the user can compress, decompress and querying XML documents.

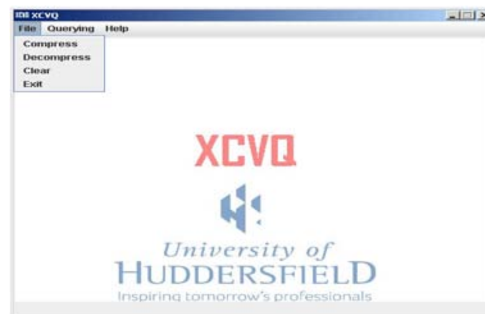


Figure B-1: The main screen of *XCVQ*

This section illustrates the implementation part of *XCVQ* compressor, decompressor, and the vague query processor.

1. Implementation of XCVQ-C

According to all the advantages of using SAX parser mentioned in section 2.2.2 to parse the given XML document, SAX parser (from *Eclipse* environment for java programming language) is used to scan the XML document. This type of parsing scans the document only once by detecting several events from that document. During each event *XCVQ-C* collects information from the document in order to use it in the compression process. The events and the work through each one are listed below and illustrated in the class diagram in Figure B-2:

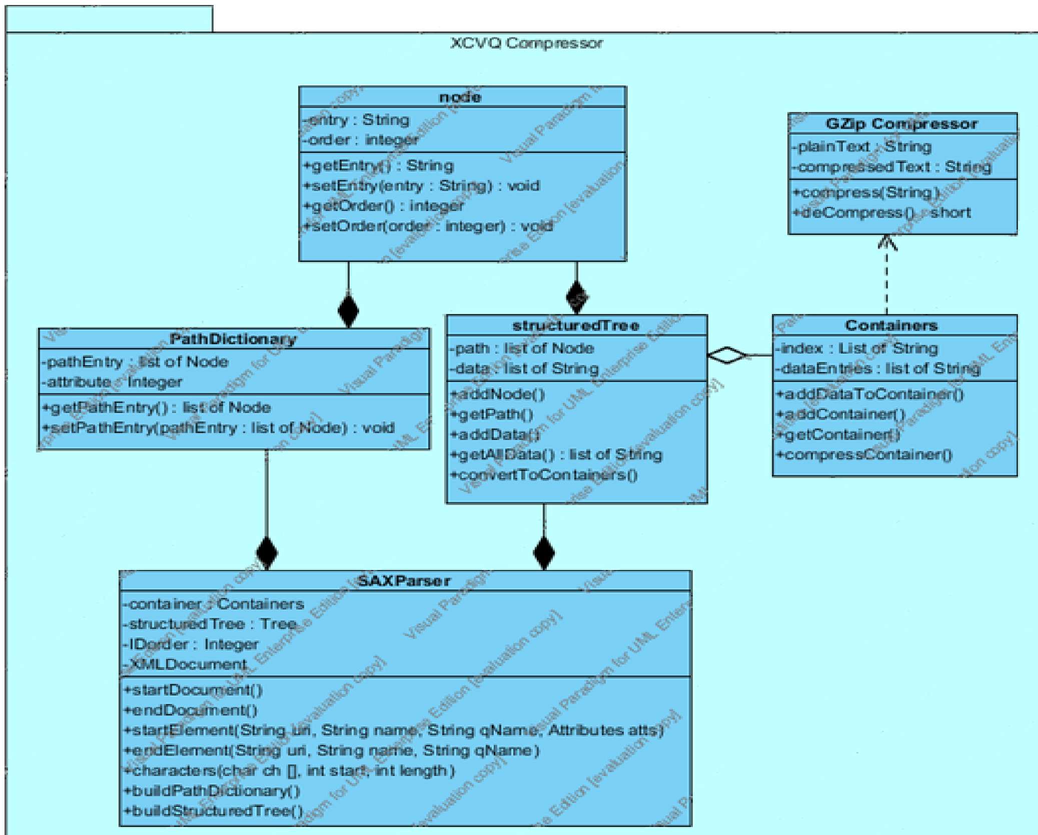


Figure B-2: *XCVQ-C* Class Diagram

4. (`StartDocument`): this event is cached only once by SAX when it detects the first tag of the document. In this stage *XCVQ-C* only initializes the used variables and prepares the used data structures and the output file to receive the data. Furthermore it specifies the name space used in that document and save it for further processes.

5. (*startElement*): this event is coached by SAX each time it detects an open tag. It holds the name of the element (*qName*) and the list of attribute names and values associated with this element (if any). In this stage, *XCVQ-C* performs the algorithm in Figure 4-6.

```
7.    Algorithm characters(chaArray ch[])
8.        data+=ch[];
9.        ignoreWhiteSpaces(data)
10.   End.
```

Figure B-4: *(character)* algorithm

6. (`chracters`): this event occurs when a data value appeared in the XML document. SAX could process this event more than once to deal with the same data. The data value is accumulated and added to the list of data in a leaf node in its appropriate path as illustrated in Figure B-4.

7. (`endElement`): SAX processes this event when it catches the end of an element, a case means that there is a piece of data ready to be inserted in a leaf node of the structured-tree (if that element holds data). The suitable path can be known from the contents of the *pathStack* as described in Figure 4-7.

8. (`endDocument`): this event is processed only once by SAX when it catches the very end of the XML document. In this stage the containers first are created from the structured-tree as illustrated in Figure 4-5. Each container has an index which represents the path from the root to the leaf for the data contained in this container. Secondly, the contents of each container are compressed using one of the back-end general purposes compression techniques either Gzip or LZW. The complete algorithm for LZW compressor is shown in Figure B-7.

```
26. Algorithm LZW(String input)
27.   input={c1, c2, …cn}
28.   let dictionary={all the 256 printable characters}
29.   lookUpString=c1
30.   for all ci ∈ input: i=2, 3, …n
31.     lookUpString=lookUpString+ ci
32.     if (lookUpString) ∉ dictionary
33.       add(lookUpString)to the end of dictionary
34.       output the position of lookUpString+ci in
          dictionary
35.       lookUpString= ci+1
36.     else
37.       lookUpString=lookUpString+ ci+1
38.       While (lookUpString + ci+1 ∈ dictionary)
39.         i++
40.         lookUpString=lookUpString+ ci
41.       Output the position of lookUpString in
          dictionary
```

Figure B-7 :*(LZW)* algorithm

The LZW algorithm starts with filling the first 256 positions in the *dictionary* with the 256 printable characters (line 3). The scanning process for the input string starts character by character in an attempt to look for the maximum sequence of characters belongs to the dictionary and add the index of this sequence to the output file (Salomon, 2007). Otherwise, if this sequence has not been added to the dictionary yet, the algorithm adds it to the end of the dictionary. For the Gzip compressor, *XCVQ-C* uses the java *(java.util.zip)* package in order to compress the required data. This package has several classes and one of them is *(GZIPOutputStream)* class

which consists of more than one constructor, each of which is used to convert the input stream into a zip file.

After implementing the compressor algorithm, some information about the compressed file is appeared to the user, as shown in Figure B-8, including the compression ratio. The new compressed file is saved with the same name and in the same path as the original XML document with (*.zip*) extension.

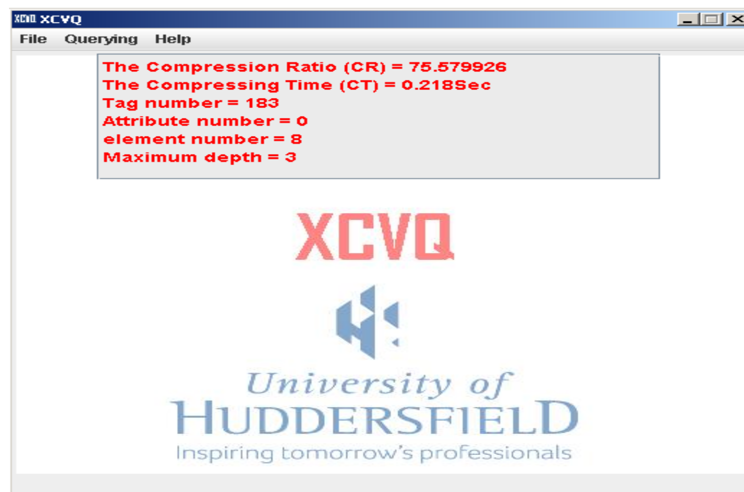


Figure B-8: GUI for compression results.

2. XCVQ-D Implementation

The implementation of the *XCVQ-D* depends restore the compressed XML document into the same containers used in the compression stage as illustrated in Figure B-9. Before applying the decompression algorithm in Figure 4-6, *XCVQ-D* decompresses the container's contents using GZip decompression technique and then uses these containers alongside with the *pathDictionary* to reproduce the XML document.

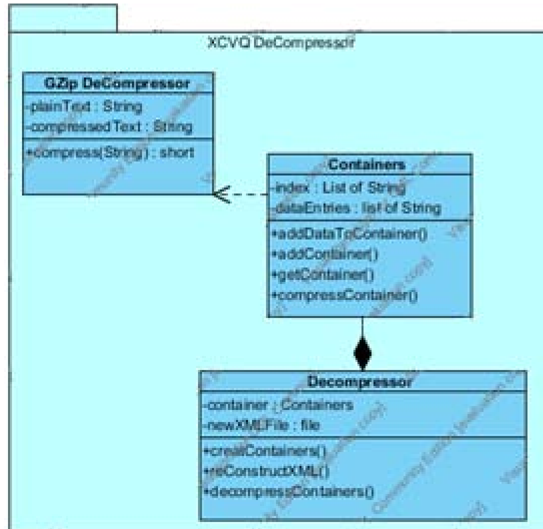


Figure B-9: *(XCVQ-D)* class diagram

The new decompressed XML document is saved in the same path as the compressed XML document, carrying its name followed by (*_1.xml*) to differentiate it from the original XML document.

3. XCVQ-QP Implementation

The implementation of *XCVQ-QP* passes through several stages. Each stage has specific roles and certain classes which are illustrated in Figure B-10.

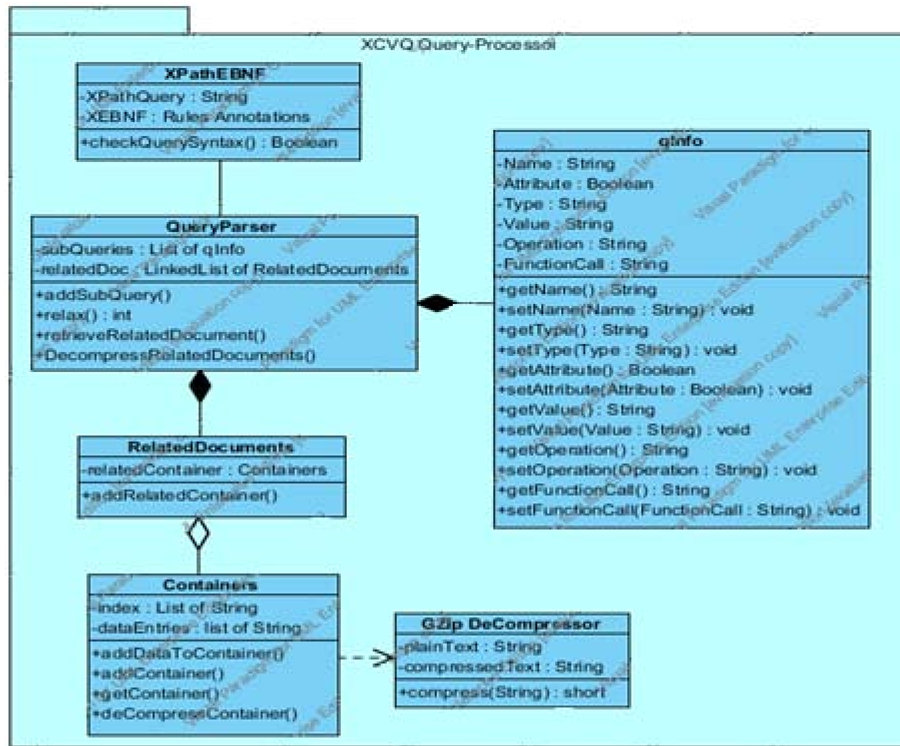


Figure B-10: *XCVQ-QP* class diagram

The main steps of each stage and the detailed roles are listed in the following sections.

a. XPath's EBNF expansion

When the user writes a vague query using the GUI in Figure B-11, this syntax of this query is checked against the XPath Extended Backus-Naur Form (EBNF) (W3C, 2007a) which specifies the grammar of XPath language. The complete EBNF for XPath query language can be seen in Appendix-A.

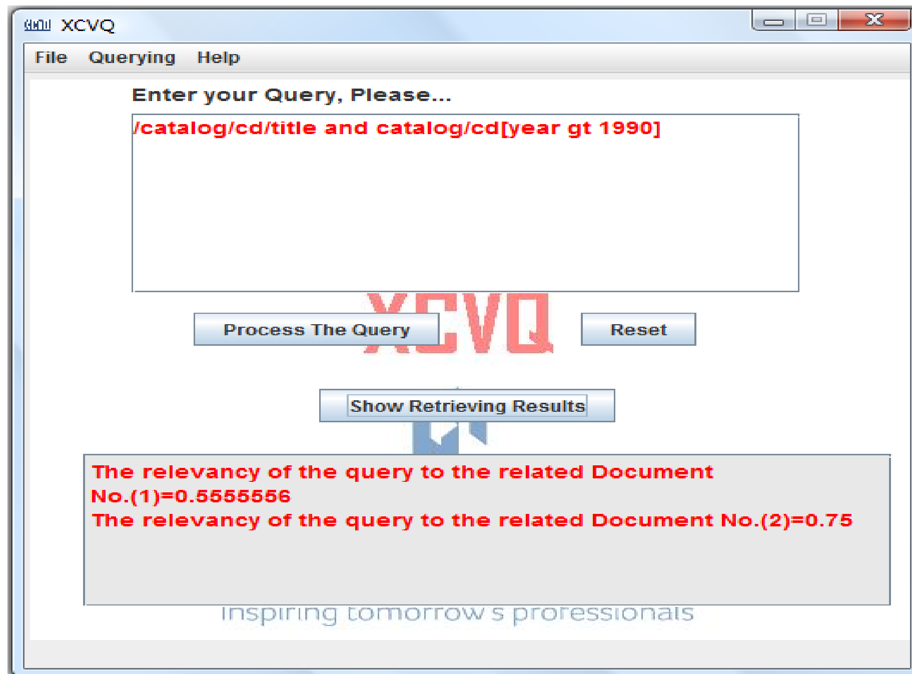


Figure B-11: GUI for *XCVQ-QP*

Since *ZXCQ-QP* performs expansion on XPath grammars to provide it the capability of accept vague conditions, an expansion process is performed on the XPath EBNF. This expansion includes:

```
PrimaryExpr   ::=  Literal | VarRef | ParenthesizedExpr
                   | ContextItemExpr | FunctionCall |
                   FunctionCallList
FunctionCallList ::= “synonyms” “(“ StrinLiteral “)”
                   | “similar(“ StrinLiteral “)”
                   | “avg(“ pathExpr “)”
                   | “median” “(“ pathExpr “)”
                   | “between” “(“ IntegerLiteral
                      | DecimalLiteral
                      | DoubleLiteral “,” IntegerLiteral
                                          | DecimalLiteral
                                          | DoubleLiteral “)”
```

Using Java Compiler-Compiler (JavaCC), the expanded EBNF for XPath is converted into executable java source code to makes it possible to follow the instructions of the EBNF and checks the syntax and lexical errors in the user's query. If the query does not meet the XPath grammar, an error message appears to the user determining the exact place of the error within the query.

The syntactically correct query is converted into a tree structure depending on the semantic of the XPath query. As an example, Figure B-12- (a) shows a query and (b) shows its semantic tree. The semantic tree for each query determines the type of each part of that query. In this example the query is divided into two main branches since it has the (AndExpr and), each branch is a (PathExpr). The first branch holds the (CATALOG, CD, and TITLE) QNames, while the second branch holds (CATALOG, CD, and Year) QNames with the (IntegerLiteral) accompanied the (YEAR) element.

The tree structure of the given query is used by *XCVQ-QP* to determine the type of each QName and to build the required data structure in order to process the query as discussed in the next section.

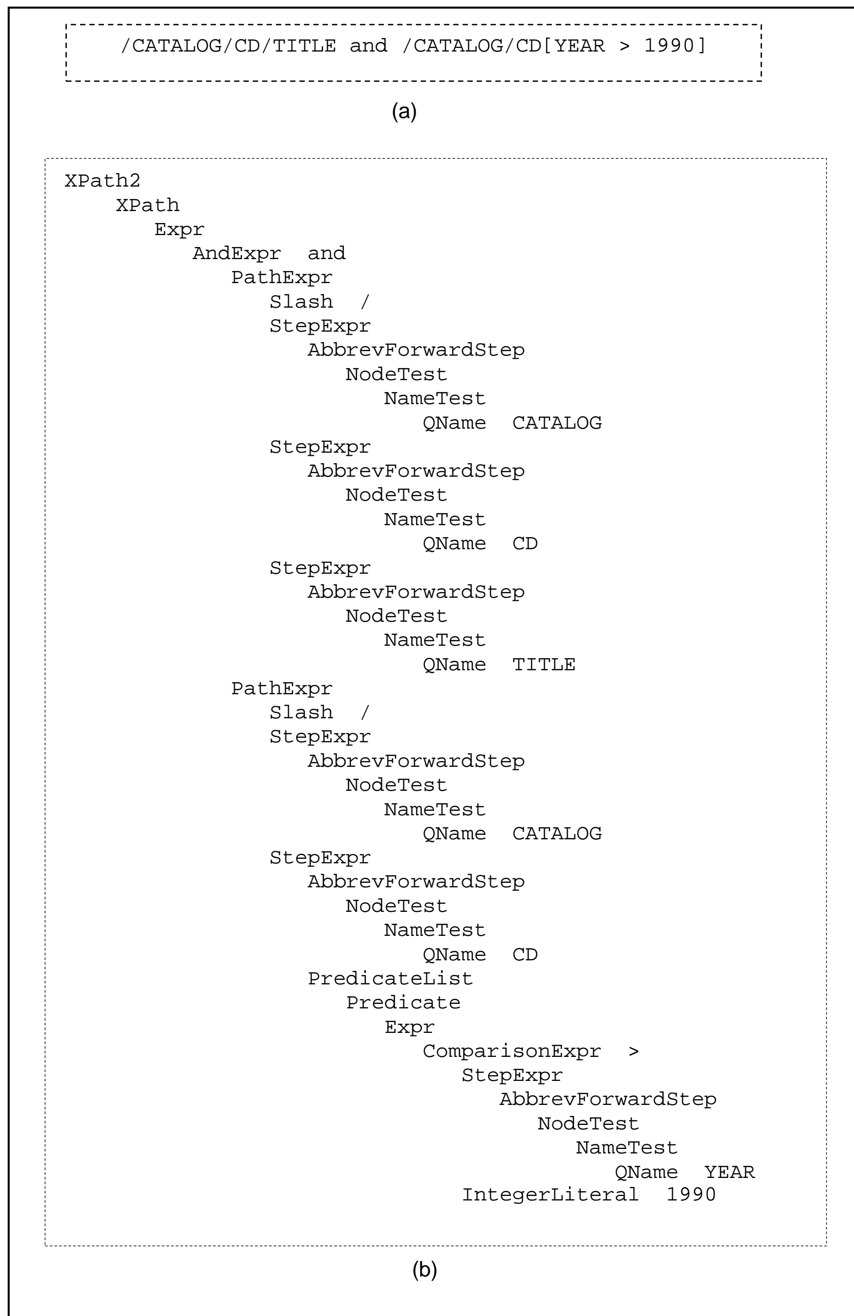


Figure B-12: An XPath query (a) and its semantic tree (b)

b. Storing the Query

XCVQ-QP uses a pre-designed data structure named (*qInfo*) in order to store all the required information from the query. This structure has the following fields:

Table B-1: The set of types provided by XCVQ-QP to *qName*

predicateLiteral	predicateString	comparisonLiteral	comparisonString
comparisonPath	functionName	andExpr	orExpr
ifVariable	forVariable	ifExpr	forExpr
pathExpr			

9. *Name*: this field stores all the *QNames* appear in the query.
10. *Attribute*: it is a Boolean field which is true if the *QName* is an attribute and is false otherwise.
11. *Type*: in this field the type of each *QName* is stored. *XCVQ-QP* provides each *QName* a specific type according to its position and role in the query. The set of provided types is shown in Table B-1. These types cover all the kinds of XPath query that can be processed by *XCVQ-QP*.
12. *Value*: if there is a literal or a string value in the query, then it is associated with the proper *qName*.

Table B-2: the *qInfo* structure for the query in Figure B-13

Name	*Attribute*	*Type*	*Value*	*Operation*	*FunctionCall*
CATALOG	False	andExpr			
CD	False	pathExpr			
TITLE	False	pathExpr			
CATALOG	False	andExpr			
CD	False	predicateLiteral			
YEAR	False	comparisonLiteral	1990	>	

13. *Operation*: this field stores the arithmetical and logical operations in the query.
14. *FunctionCall:* stores the list of parameters for the function if there is one in the query.

Table B-2 illustrates the *qInfo* structure for the query example in Figure B-12.

```
44.  Algorithm processXPathQuery(structure qInfo)
45.  let qInfo= [a1, a2, …, an] such that:
46.  let fileDB=[f1, f2, …, fm]
47.  let relevant=[r1, r2, …, rk]
48.  for all ai.Name ∈ qInfo
49.    for all fj ∈ fileDB
50.       if ai.Name ∈ fj.pathDictionary
51.          relevant.pathDictionary ← fj
52.          relevant.query ← ai
53.  for all ri ∈ relevant
54.      subQuery= Decpmpose(ri.query)
55.      cost=relax(ri.subQuery)
56.      if cost>threshold
57.         remove (ri) from relevant
58.      tree=retrieveData
59.  allTrees=Combine all the retrieved trees
```

Figure B-13: Processing an XPath query algorithm.

c. Query decomposition: Stage-1

After collecting the important information from the query and store them in (*qInfo*) structure, the algorithm in Figure B-13 is processed. This algorithm starts by collecting the relevant documents from the compressed XML repository. During this process each compressed document is scanned for one of the *Name* field in the *qInfo* structure, if it contains one of them in its *path-dictionary*, then this file is candidate to be one of the relevant documents. If *(n)* relevant document encountered, only the *path-dictionary* for these documents are loaded into the memory and the original *qInfo* structure is decomposed into *(n)*

structures each of which is associated with its relevant document (lines 5-9). This process decomposes the XPath query into several sub-queries according to their relevancy to a specific document.

d. Query Decomposition: Stage-2

After completing the first stage decomposition, the second stage of the decomposition is started by applying the decomposition function in Figure B-14 on each sub-query to produce a new set of sub-queries (line11 in Figure B-13). This function determines the relevant containers from the compressed file. This is done through checking if any *Name* field of the given sub-query is contained within the index of that container. In this case the relevant container is uploaded into the memory alongside with its relevant part from the sub-query.

```
1. Function decompose (query Q)
2. let Containers=[c1, c2, …, cn]
3. For all ci
4.     If Q.namej ∈ ci.index
5.         newSub-queries ← Q.namej
6.         newSub-query ← ci
7.     }
8. End.
```

Figure B-14: Query decomposition function

For all the new sub-queries, each one is relaxed against the index of its relevant container, (line 12). This relaxation is done by performing a matching process between the sub-query and the index of its relevant container. This process performs changes on the original sub-query in order to fit it with the index by adding, removing, renaming, or reordering position in the nodes of the query. After each change, the cost of that change is computed and the total cost of relaxation is checked against a pre-defined threshold to determine if this

query should be removed from the list of relevant queries if it has high cost (lines 13-14).

e. Retrieving and combining results

At this stage, (line 17), each container accompanied with its sub-query is on the memory ready to be retrieved. The retrieving process taking into consideration the type of each *qName* and its operation and retrieve only the required information. If the query required retrieving information from the data set attached in the container, decompression process is performed only on this single container in order to retrieve the required data. The example query in Figure B-12 has a predicate requiring the values of the (*YEAR*) data to be greater than (*1990*) which requires the performance of the decompression only on one container that has the data and filter these data to retrieve only the data that meet the condition.

Until this stage no decompression required when the query is structured based one. After combining all the retrieved sub-documents, each one is decompressed, using the same decompression algorithm in Figure 4-6, and combined under one XML document to form single tree instead of a forest.

The resulted document can be compressed again if the user needs to make further querying on it.

Appendix-C: XML Corpus & XPath Benchmark

This appendix contains the XML corpus that were used in the testing process (Table C-1), the description of its groups, and the complete XPath benchmark that were used in the testing process (Table C-2).

Table C-1: XML Corpus

Group#	XML file name	File Size/MB	%SR	Tag no	Element no	Attributes no	Max depth
1	321gone	2.441E-2	38.06	311	32	0	6
	Ebay	3.515E-2	11.14	156	32	0	6
	Ubid	2.050E-2	42.63	342	32	0	6
	Yahoo_Shopping	2.539E-2	34.1	342	32	0	6
	Homeseekers	2.603	58.12	59322	35	0	5
	Nky	3.213	70.43	112051	50	0	5
	Texas	3.177	58.7	84577	54	0	5
	Yahoo_Homes	0.419	56.77	11038	33	0	3
	XMark-1	11.325	30.16	520546	74	0	12
	XMark-2	113.061	30.03	5167121	74	0	12
	(Schmidt et al., 2002; Washington, 2002)						
2	Baseball (Washington, 2002)	0.632	92.95	28306	46	0	6
3	Berkeley	9.277E-2	32.96	1143	15	0	6
	Cornell	3.027E-2	45.64	833	15	0	6
	Michigan	6.738E-2	46.68	1899	15	0	6
	Texas	3.222E-2	44.88	859	15	0	6
	Washington	5.175E-2	33.28	1025	15	0	6
	Read	0.283	63.22	10546	16	0	5
	Uwm	2.226	58.41	66729	16	6	6
	Wsu	1.558	73.99	74557	16	0	5
	(Washington, 2002)						
4	CD-Catalog	0.598	63.68	183	8	0	3
5	DBLP (Washington, 2002)	131.167	45.1	4718588	32	3	5
6	EnWikiNews	69.421	10.13	2103778	20	4	6
	EnWikiQuote	124.532	3.69	2672870	20	4	6
	EnWikiVersity	81.397	10.64	3333622	20	4	6
	EnWikTionary	556.612	26.26	28656178	20	4	6
	(Wikipedia, 2001)						

7	**EXI-Array**	**22.062**	**43.7**	**226523**	**47**	**0**	**14**
	EXI-factbook	**4.042**	**47.47**	**55453**	**199**	**0**	**8**
	EXI-GeogCoord	**15.828**	**0.003**	**17**	**30**	**0**	**11**
	EXI-Invoice	**0.934**	**65.41**	**15075**	**52**	**28**	**9**
	EXI-weblog	**2.526**	**72.49**	**93435**	**12**	**0**	**3**
	(EXI, 2009)						
8	**GB-meta**	**48.82**	**71.32**	**886419**	**97**	**18**	**13**
	Sweden-meta	**3.35**	**71.14**	**60614**	**101**	**17**	**12**
	Turkey-meta	**5.85E-3**	**73.66**	**100**	**48**	**2**	**8**
	(European, 2003)						
9	**Henry IV, Part I**	**0.274**	**34.21**	**4334**	**14**	**0**	**7**
	Richard II	**0.251**	**32.48**	**4116**	**16**	**0**	**8**
	j_caesar	**0.181**	**37.95**	**4455**	**16**	**0**	**8**
	Shakespeare	**7.529**	**36.56**	**179690**	**22**	**0**	**9**
	(plays, 2000)						
10	**LineItem**	**30.799**	**83.47**	**1022976**	**18**	**0**	**3**
	XBench-DCSD-Normal	**105.368**	**57.24**	**2242699**	**50**	**0**	**10**
	XBench-DCSD-Small	**10.578**	**57.3**	**2259292**	**50**	**0**	**10**
	(Washington, 2002; Waterloo, 2003)						
11	**Mondial (Washington, 2002)**	**1.778**	**48.74**	**22423**	**23**	**45**	**8**
12	**NASA (Washington, 2002)**	**24.622**	**37.13**	**476646**	**61**	**0**	**11**
13	**SwissPort (Washington, 2002)**	**112.761**	**56.5**	**13917441**	**85**	**0**	**5**
14	**Tree Bank (Washington, 2002)**	**85.416**	**31.65**	**10795711**	**250**	**0**	**36**

The selected XML documents in the corpus were organized into many groups according to their origins and the purpose of their use, as follows:

Group-1: It consists of many XML documents that are used in online shopping processes through different e-shopping and auction web sites. These documents are converted from database systems and they contain many empty elements with neither data nor sub-elements inside them.

Group-2: the XML document in this group provides a complete description to all the teams including all the details about their players who participated in 1998 national league.

Group-3: This group contains XML documents from different academic department. Some of the documents describe simple CVs for the academic staff

in these departments and the courses they teach. The other documents describe the detailed information for the courses submitted by some academic departments in different universities.

Group-4: The Document in this group gives details about many songs CDs such as their name, publication year, and their country.

Group-5: This group has only one document that illustrates different papers published in proceeding of conferences and journals in the field of computer science.

Group-6: different backup documents from Wikipedia web site are collected in this group.

Group-7: This group contains sample documents from a collection of documents collected by the Efficient XML Interchange (EXI) working group which is part of the W3C. These documents contain the needed information in data exchanging.

Group-8: The XML documents that describe the detailed climate changes in different countries around the world are listed in this group.

Group-9: This group has some of Shakespeare's plays which considered being (TD) document type.

Group-10: This document contains a huge amount of shipping information for online shopping for different items taken from Google web site.

Group-11: The XML document in this group contains lots of statistical information about many countries around the world such as their population, area, available natural resources, etc.

Group-12: This document is transferred from NASA database which includes summarization of some of the NASA projects converted from text file.

Group-13: The complete description of the DNA sequence is described in the XML document in this group.

Group-14: This document contains many parsed and encrypted English sentences taken from the Wall Street Journal.

Table C-2: XPath Query Benchmark

QFT Concept	Query Name	Query	Description
Axes	*Q1*	**/catalog/cd**	**normal path (exact matching)**
	Q2	**/catalog/title/cd**	**Path out of order**
	Q3	**/cd/year/catalog**	**Does not start from the root**
	Q4	**/catalog/title**	**a gap exists , the actual path is (/catalog/cd/title)**
	Q5	**/catalog/yeer**	**miss spelling in the element name**
	Q6	**/cd/cateloge/yeer**	**miss spelling in more than one element name**
	Q7	**/catalog/cd/year/title**	**Sibling elements(year, title)**
Filters	*Q8*	**/catalog/cd/year[5]**	**Normal partial match (position filter)**
	Q9	**/catalog/cd/year["1990"]**	**Normal partial match (value filter)**
	Q10	**/cd/catalog/country["uk"]**	**Out of order path + predicate**
	Q11	**/cd/title/year/country[8]**	**Siblings + predicate**
	Q12	**cataloge/yeer/cuntry["USA"]**	**Spelling errors + predicate**
Operators	*Q13*	**/catalog/cd[year lt 2000]**	**Normal predicate with comparative operator**
	Q14	**/cd/title/artist[year ge 1990]**	**Sibling + comparative operator**
	Q15	**/cd/title[year lt 1990][country eq "uk"]**	**More than one comparative operator**
	Q16	**/cd/title/country["uk"][yeer le 1990]**	**Predicate + comparative operator**
	Q17	**/cd/title/year eq 1990**	**comparative operator without predicate**
	Q18	**/cd/year=2000**	**Relational operator (the result is either True or False)**
	Q19	**cd/title/yeer !=1998**	**Siblings + miss spelling + operation**
	Q20	**/cd[year gt 1990] and /cd[country eq "uk"]**	**(and) operator**
Functions	*Q21*	**/cd/title/synonyms("date")**	**Find the synonyms of an element name**
	Q22	**cd/country eq synonyms("Britain")**	**Find the synonym of a data value**
	Q23	**/cd/title/similar(artest)**	**Find the similar element name**

	Q24	**/cd/similar(artest)/title**	**Find the similar element name**
	Q25	**/cd/year/title eq similar("keep your heart")**	**Find the similar data value**
	Q26	**/cd/artist/count(title)**	**Find the number of occurrences of an element**
Multi-File	*Q27*	**/cd/book/year/title**	**Exact match**
	Q28	**/cd/book/title/artist/author/year**	**Siblings**
	Q29	**book/title/cd/yeer/aother**	**Miss spelling**
	Q30	**/cd/book/title/year[“1990”]**	**Data value predicate**
	Q31	**book/title/year[4] and /cd/title/year**	**Order predicate**
	Q32	**/cd/book[year lt 1990][country eq "uk"]**	**Multiple predicates**

Appendix-D: Testing Results

The complete set of data that had been used to test and evaluate *XCVQ-C* and *XCVQ-D* is listed in the following table. This table contains all the actual results for these testing.

XML file name	SCR	SCT/Sec	SDC/Sec	CR	CT/ Sec	DT/Sec
Turkey-meta	48.02	0.031	0.47	78.17	0.047	0.32
Ubid	60.34	0.032	0.31	83.35	0.047	0.47
321gone	38.99	0.042	0.46	73.31	0.078	0.47
Yahoo_Shopping	42.47	0.051	0.62	76.54	0.087	0.64
Cornell	68.66	0.062	0.31	87.5	0.102	0.47
Texas	45.71	0.063	0.32	81.75	0.13	0.78
Ebay	16.51	0.031	0.48	70.29	0.163	0.75
Washington	50.55	0.031	0.78	81.1	0.2	0.63
Michigan	70.83	0.047	0.41	89.3	0.42	1.09
Berkeley	50.81	0.094	0.47	81.94	0.538	0.62
j_caesar	30.32	0.14	0.64	70.91	0.58	1.1
Richard II	24.9	0.22	0.8	68.51	0.72	1.56
Henry IV, Part I	26.85	0.37	1.59	68.8	0.85	1.1
Read	64.82	0.583	1.85	87.33	0.988	1.14
Yahoo_Homes	43.32	0.68	3.04	83.35	0.96	2.19
CD-Catalog	57.67	0.77	4.13	75.53	0.94	2.32
Baseball	62.19	0.95	5.53	83.56	1.078	2.34
EXI-Invoice	58.98	0.98	5.62	73.63	1.28	3.12
Wsu	64.81	21.85	20.62	87.35	1.56	3.59
Mondial	59.37	14.64	32.81	85.35	4.42	6.56
Umw	66.37	39.17	31.56	90.35	4.43	4.37
EXI-weblog	44.6	54.8	57.81	72.38	7.3	4.84
Homeseekers	47.9	36.98	72.75	86.21	12.6	32.41
Texas	67.24	42.93	79.7	86.8	16.75	36.04
Nky	50.75	29.54	92.1	83.69	12.3	36.4
Sweden-meta	78.23	57.53	97.6	93.52	12.4	36.41
EXI-factbook	37.5	61.71	103.94	74.2	80.8	29.07
Shakespeare	26.01	85.5	148.6	69.32	130.8	32.09
XBench-DCSD-	27.76	103.7	155.7	69.92	194.6	38.28

Small						
XMark-1	38.45	120.8	165.3	75.38	225.6	39.01
EXI-GeogCoord	0.0031	169.1	198.8	75.61	374.7	40.1
EXI-Array	22.062	210.9	226.4	72.56	458.5	43.4
NASA	39.18	256.3	254.2	88.51	517.3	43.9
LineItem	30.799	312.8	297.7	81.51	692.4	78.6
GB-meta	77.22	370.6	337.2	75.67	734.2	81
EnWikiNews	69.421	398.9	412.3	68.52	810.6	89.89
EnWikiVersity	81.397	417.7	489.8	70.55	894.7	97.39
Tree Bank	37.68	426.3	504.4	79.8	956.2	99.87
XBench-DCSD-Normal	30.19	486.9	645.6	71.38	1069.8	100.19
SwissPort	58.12	502.6	702.4	81.2	1296.4	105.75
XMark-2	37.89	524.5	826.9	77.02	1368.8	113.65
EnWikiQuote	68.25	565.8	924.3	69.69	1438.1	123.9
DBLP	68.16	605.1	1022.3	79.54	1578.7	157.45
EnWikTionary	85.43	1104.7	3750.7	70.85	4152.7	260.9

Appendix-E: XPath Query Evaluation Benchmark

This appendix contains the complete set of queries that had been used to evaluate *XCVQ-QP* and comparing the results with other queriable XML compressors. It consists of three sets of queries:

Set-1**:** The queries listed in this set were used to test the performance of XGrind and Xpress compressors, and were used to evaluate *XCVQ-QP* and compare the results with these two techniques.

XML document	Query Name	Query
BaseBall	B1	*SEASON/LEAGUE/DIVISION/TEAM/PLAYER/GIVEN NAME*
	B2	*//TEAM/PLAYER/SURNAME*
	B3	*/SEASON/LEAGUE//TEAM/TEAM CITY*
	B4	*/SEASON/LEAGUE//TEAM[TEAM CITY >= Chicago and TEAM CITY <= Toronto]*
Umw	C1	*/root/course/selection/session/place/building*
	C2	*//session/time*
	C3	*/root/course//session/time/start time*
	C4	*/root/course//session/time[start time >= 800 and start time <= 1200]*
Shakespeare	S1	*/PLAY/ACT/SCENE/SPEECH/STAGEDIR*
	S2	*//PGROUP/PERSONA*
	S3	*/PLAY/ACT//SPEECH/SPEAKER*
	S4	*/PLAY/ACT//SPEECH[SPEAKER>= CLEOPATRA and SPEAKER <= PHILO]*

Set-2: this set of queries, listed in the following table, was used to evaluate *XCVQ-QP* against XQzip compressor.

XML document	Query Name	Query
LineItem	L1	***//table/T/L_TAX***
	L2	***/table/T[L_TAX = "0.02"]***
	L3	***/table/T[L_TAX[[. >= "0.02"]]]***
	L4	***//T[L_ORDERKEY = "100"]***
	L5	***//L_ DISCOUNT***
TreeBank	T1	***//_QUOTE_//_NONE_***
	T2	***//_QUOTE_//_BACKQUOTES_***
	T3	***//_QUOTE_//NP[_NONE_ = "FTTVhQZv7pnPMt+EeoeOSx"]***
	T4	***//_QUOTE_//SBAR//VP/VBG***
	T5	***//_QUOTE_//NP/PRP_DOLLAR_***
Shakespeare	S1	***//SPEAKER***
	S2	***//PLAY//SCENE//STAGEDIR***
	S3	***//SPEECH[SPEAKER = "PHILO"]/LINE***
	S4	***//SCENE/SPEECH/LINE***
	S5	***//SCENE[TITLE="SCENE II. Rome. The house of EPIDUS"]/LINE***

Set-3: the queries listed in the following table were used to evaluate *XCVQ-QP* against XSAQCT compressor.

XML document	Query Name	Query
dblp	D1	***/dblp/article/cdrom***
	D2	***/dblp/mastersthesis/@key***

LineItem	L1	***/table/T/L_COMMENT***
	L2	***/table/T/L_ORDERKEY***
Shakespeare	S1	***/PLAYS/PLAY/TITLE***
	S2	***PLAYS/PLAY/ACT/SCENE/STAGEDIR***
SwissPort	P1	***/root/Entry/@id***
	P2	***/root/Entry/Ref/Comment***
uwm	U1	***/root/course_listing/course***
	U2	***/root/course_listing/restrictions/A/@HREF***

Appendix-F: Independent testing

The following table contains the XML documents that have been used in an independent testing to find the compression ratio using *XCVQ*. To find the compression ratio, the following equation was used:

$$\textbf{Compression ratio} = \frac{\textbf{Original}_{\textbf{XML}}\textbf{file size} - \textbf{Compressed}_{\textbf{ZIP}}\textbf{file size}}{\textbf{Original}_{\textbf{XML}}\textbf{file size}}$$

The independent testing was made on environment Quad-Core Intel Xeon processor that has the speed of 2.8 GHz. The operating system was Mac OS X 10.6.4 with 8GB of hard drive.

XML File Name	Compression Ratio
Setup of points	50%
books1	50%
cd_catalog	60%
TURKY_meta	66.7%
data_20101111102811	57.9%
ubid	81%
321gone	0.72%
yahoo	73.1%
cornell	87.1%
texas	84.8%
ebay	69.4%
washington	81.1%
berkeley	81.4%
j_caesar	71%
rich_ii	68%
Hen_vi_1	68.7%
reed	87.2%
yahoo_homes	83.3%
BaseBall	83.4%
EXI-Invoice	79.8%
Mondial	85.3%
uwm	90.35%
EXI-weblog	88.3%
homeseekers	86.2%
texas_house	81.7%
nky.xml	83.7%
SWEDEN_meta	93.6%
EXI-factbook	81.5%

Appendix-G: XML dummy elements ratio

The following table lists the ratio of the dummy elements in the tested XML documents which is the same ratio that represents the loss in the structure of these documents.

XML file name	%Dummy elements ratio
321gone	7.4
Ebay	1.3
Ubid	16.8
Yahoo_Shopping	9.2
Homeseekers	6.2
Nky	3.8
Texas	6.3
Yahoo_Homes	2.2
XMark-1	1.5
XMark-2	0.4
Baseball	1.2
Berkeley	12.2
Cornell	19.3
Michigan	19.5
Texas	4.1
Washington	4.7
Read	4.3
Umw	14.6
Wsu	12.4
CD-Catalog	0.0
DBLP	0.0
EnWikiNews	0.9
EnWikiQuote	0.3
EnWikiVersity	1.2
EnWikTionary	1.1
EXI-Array	0.0
EXI-factbook	0.0
EXI-GeogCoord	0.0
EXI-Invoice	0.0
EXI-weblog	0.0
GB-meta	0.0
Sweden-meta	5.9
Turkey-meta	0.0
Henry IV, Part I	1.6
Richard II	1.4
j_caesar	1.2
Shakespeare	2.4
LineItem	0.0
XBench-DCSD-Normal	0.0
XBench-DCSD-Small.xml	0.0

Mondial	**9.2**
NASA	**0.0**
SwissPort	**0.0**
Tree Bank	**0.0**