UNIVERSIDADE DA CORUÑA

Departamento de Computación

Compressed Self-Indexed XML Representation with Efficient XPath Evaluation

Tesis Doctoral

Doctoranda: Ana Belén Cerdeira Pena

Directores: Nieves Rodríguez Brisaboa, Gonzalo Navarro Badino

A Coruña, Enero de 2013

PhD thesis supervised by
Tesis doctoral dirigida por

Nieves Rodríguez Brisaboa
Departamento de Computación
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
brisaboa@udc.es

Gonzalo Navarro Badino
Departamento de Ciencias de la Computación
Universidad de Chile
Blanco Encalada 2120 Santiago (Chile)
Tel: +56 2 6892736
Fax: +56 2 6895531
gnavarro@dcc.uchile.cl

A mis padres y hermana

Acknowledgments

My first gratitude words are addressed to my thesis advisors, Nieves and Gonzalo. You both trusted in me from the beginning, and showed me that with your complete dedication whenever I needed, and your constant support throughout this work. You helped me to follow the right way with your knowledge and experience. Today this thesis is the result of all what you have taught me along this process. Thank you for your faith and unconditional help.

I also want to give thanks to all members of the Database Laboratory, working mates, but also friends, who make me enjoy every day the hours of work. With special affection, I express my gratitude to Antonio Fariña and Luisa Carpente, for their invaluable help, and constant encouragement. It would had not been the same without your support.

I can not forget all people I met during my research stays far from home, because of their hospitality, and friendship, when one most needs it. In particular, I would like to acknowledge to Diego Arroyuelo, Francisco Claude, Rodrigo Paredes, Rodrigo Cánovas, Daniel Valenzuela, Mauricio Marín, Miguel A. Martínez, Gabriella Pasi, and Emanuele Panzeri. I also want to give special thanks to Felipe Sologuren and Kim Nguyen, for their unselfish help in research topics.

Thanks as well to you, Miguel, for your enormous patience, and for all those years we have shared together. They have been wonderful, and I know they will remain so.

Finally, and undoubtedly, my biggest gratitude is for my family. My parents and sister. You have always been by my side, sharing with me the joys, but also suffering in the bad moments. Whenever I felt down, you gave me the strength to continue, because you taught me that dreams can not be achieved without effort, without fighting for them. I've followed your example, and today one of my dreams has come true. You were right, it has not been easy, but the fact is that what has been difficult, would have been impossible without you.

Agradecimientos

Mis primeras palabras de agradecimiento van dirigidas a mis directores de tesis, Nieves y Gonzalo. Creísteis en mí desde el primer momento, y así me lo demostrasteis con vuestra completa dedicación, siempre que necesité de vuestros consejos, y vuestro apoyo constante a lo largo de todo este trabajo. Me habéis ayudado a seguir el camino correcto con vuestros conocimientos y experiencia. Hoy, esta tesis es el resultado de todo aquello que me habéis enseñado a lo largo de este proceso. Gracias por vuestra confianza y ayuda incondicional.

También quiero dar las gracias a todos los miembros del Laboratorio de Bases de Datos, compañeros de trabajo, pero también amigos, que hacen que cada día disfrute de las horas de trabajo. Con especial cariño, doy las gracias a Antonio Fariña y Luisa Carpente por su ayuda inestimable y ánimos constantes. No habría sido lo mismo sin vuestro apoyo.

No puedo olvidarme tampoco de todas aquellas personas que he ido conociendo durante mis largas estancias fuera de casa, por su hospitalidad y amistad, cuando uno más lo necesita. En particular, me gustaría mencionar a Diego Arroyuelo, Francisco Claude, Rodrigo Paredes, Rodrigo Cánovas, Daniel Valenzuela, Mauricio Marín, Miguel A. Martínez, Gabriella Pasi y Emanuele Panzeri. También agradezco especialmente a Felipe Sologuren y Kim Nguyen su ayuda desinteresada en temas de investigación.

Gracias además a ti, Miguel, por tu inmensa paciencia y haber compartido conmigo todos estos años. Han sido maravillosos, y sé que lo seguirán siendo.

Finalmente, y como no podía ser de otra forma, mi mayor agradecimiento es para mi familia. A mis padres y hermana. Sois quienes habéis estado a mi lado en todo momento, compartiendo alegrías, pero también sufriendo conmigo en los malos momentos. Si me caía, ahí estabais vosotros para ayudarme a levantar, porque me habéis enseñado que los sueños sólo se consiguen con esfuerzo, luchando por ellos. He seguido vuestro ejemplo, y hoy he cumplido uno de ellos. Teníais razón, no ha sido fácil, pero lo cierto es que sin vosotros lo difícil hubiese sido imposible.

Abstract

The popularity of the *eXtensible Markup Language* (XML) has been continuously growing since its first introduction, being today acknowledged as the *de facto* standard for semi-structured data representation and data exchange on the World Wide Web. In this scenario, several query languages were proposed to exploit the expressiveness of XML data, as well as systems to provide an efficient support. At the same time, as research in compression became more and more relevant, works also focused their efforts on studying new approaches to provide efficient solutions, using the minimum amount of space. Today, however, there is a lack of practical available tools that join both efficient query support, and minimum space requirements.

In this thesis we address this problem, and propose a new approach for storing, processing and querying XML documents in time and space efficient way, by specially focusing on XPath queries. We have developed a new compressed self-indexed representation of XML documents that obtains compression ratios about 30%-40%, over which a query module providing efficient XPath query evaluation has also been developed. As a whole, both parts make up a complete system, we called XXS, for the efficient evaluation of XPath queries over compressed self-indexed XML documents. Experimental results show the outstanding performance of our proposal, which can successfully compete with some of the best-known solutions, and that largely outperforms them in terms of space.

Resumen

La popularidad del *eXtensible Markup Language* (XML) no ha hecho sino más que ir en aumento desde su introducción inicial, siendo hoy día reconocido como el estándar *de facto* para la representación de datos semi-estructurados, y el intercambio de datos en Internet. Bajo este escenario, son varios los lenguajes de consulta que se han venido proponiendo para explotar la expresividad de los datos en formato XML, así como sistemas que proporcionasen un soporte eficiente a ellos. Al mismo tiempo, y conforme la investigación en compresión se ha hecho cada vez más relevante, los esfuerzos se han dirigido también a estudiar nuevas aproximaciones que ofreciesen soluciones eficientes, pero usando además la menor cantidad de espacio posible. Actualmente, sin embargo, existe una clara ausencia de herramientas prácticas disponibles que aúnen ambas características: un soporte a la realización de consultas eficiente, con requisitos de espacio mínimos.

En esta tesis abordamos ese problema, y proponemos una nueva solución para el almacenamiento, procesamiento y consulta de documentos XML, eficiente en tiempo y en espacio, centrándonos, en particular, en el lenguaje de consulta XPath. Así, hemos desarrollado una nueva representación comprimida y auto-indexada de documentos XML, que obtiene ratios de compresión del 30%-40%, y sobre la cual se ha creado un módulo de consulta para la eficiente evaluación de consultas XPath. En conjunto, ambas contribuciones conforman un sistema completo, que hemos dado en llamar XXS, para la evaluación eficiente de consultas XPath sobre documentos XML comprimidos y auto-indexados. Los resultados experimentales evidencian el destacado comportamiento de nuestra herramienta, que es capaz de competir exitosamente con algunas de las soluciones más conocidas, a las que además supera claramente en términos de espacio.

Resumo

A popularidade do *eXtensible Markup Language* (XML) non fixo máis que medrar dende a súa introdución inicial, sendo recoñecido hoxe en día como o estándar *de facto* para a representación de datos semi-estruturados e o intercambio de datos na Rede. Baixo este escenario, son varias as linguaxes de consulta que se propuxeron para explotar a expresividade dos datos en formato XML, así como sistemas que proporcionasen un soporte eficiente a eles. Ó mesmo tempo, e conforme a investigación en compresión se fixo cada vez máis relevante, os esforzos tamén foron dirixidos a estudiar novas aproximacións que ofrecesen solucións eficientes, pero usando ademáis a menor cantidade de espacio posible. Actualmente, sen embargo, existe unha clara ausencia de ferramentas prácticas dispoñibles que agrupen ambas características: un soporte á realización de consultas eficiente, xunto con requisitos de espacio mínimos.

Nesta tese abordamos ese problema, e propoñemos unha nova solución para o almacenamento, procesamento e consulta de documentos XML, eficiente tanto en tempo como en espacio, centrándonos, en particular, na linguaxe de consulta XPath. Así, desenvolvimos unha nova representación comprimida e auto-indexada de documentos XML, que obtén ratios de compresión en torno ó 30%-40%, e sobre a cal se creou tamén un módulo de consulta para a eficiente evaluación de consultas XPath. En conxunto, ambas contribucións conforman un sistema completo, que chamamos XXS, para a evaluación eficiente de consultas XPath sobre documentos XML comprimidos e auto-indexados. Os resultados experimentais amosan o destacado comportamento da nosa ferramenta, que é capaz de competir exitosamente con algunhas das solucións máis coñecidas, ás que ademáis supera claramente en termos de espacio.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Since its first introduction in 1998, the importance of the *eXtensible Markup Language* (XML) [XMLa], has been constantly increasing, mainly due to its suitability for data exchange on the World Wide Web. Nowadays, it is widely employed and it has been acknowledged as the *de facto* standard for semi-structured data representation, being used to store large volumes of information from different domains, such as e-commerce and business, digital libraries, catalogs, chemical and biological areas, metadata specifications, and so on.

To exploit the expressive power of XML, query languages like XPath [XPaa] and XQuery [XQu] have been defined, allowing constraint formulation on both document content and structure. Their growing interest, and also the challenge of solving those query languages, have triggered much research aimed to provide efficient solutions, either as theoretical proposals or in the form of real systems. These systems are usually divided into two different categories: those that follow a *streaming* approach (such as GCX [SSK07], SPEX [SPE], etc.), hence having to sequentially read the document to answer each query; and the *indexed* ones (such as Saxon [Kay08], Galax [FSC$^+$03], MonetDB/XQuery [BGvK$^+$06], Qizx/DB [Qiz], etc.), requiring a first preprocessing of the document to build additional data structures over it, which are then used to solve the queries without sequentially traversing the whole document.

Indexed systems are very interesting solutions for many scenarios, such as those where the documents are so large that a sequential scan is prohibitively costly or when many queries must be performed over the same document. However, while *streaming* approaches are supposed to be slower than *indexed* ones, this may not always be the case. Note that indexed solutions improve querying capabilities at the expense of increasing the space requirements, due to the index structures. Thus, in case that the space needed for the index made it necessary to manipulate it on disk, efficiency could be affected by I/O transfer times. Hence many efforts have been devoted to address the problem of creating an in-memory index, and also to cope with the usual high space requirements of the *indexed* alternatives. These efforts involve the use of compression techniques to minimize that extra space.

Related to the space challenge, another quite active line of research has been the development of XML compression methods. One of the main features of the XML data model is its great flexibility. However, it also constitutes one of its main drawbacks, since the *verbosity* of XML documents may result into huge size documents, which have to be transmitted, stored and, as just seen, also queried. In this way, the use of compression tools not only saves storage space, but also time. Time is the critical factor in efficiency, and working with a compressed version of a document saves time when it is transmitted through a network, when we need to access to disk looking for a document, or more importantly, when it is processed. Therefore, compression is clearly more convenient.

Several works have been devoted in the last years to the XML compression task, both in the form of general text compressors, known as *XML-blind* compressors (e.g. Ziv-Lempel techniques [ZL77, ZL78, Wel84], Huffman compression [Huf52, dMNZBY00], PPM based methods [CW84], Dense Codes compressors [BFNP07], etc.), or compressors specifically designed to exploit XML document structure. Indeed, most of these *XML conscious* compressors have gone one step beyond, and have faced both problems, compression and query support, leading to several queriable compression tools (e.g. XGrind [TH02], XPRESS [MPC03], XCQ [LNWL03, NLWL06], XQzip [CN04], XQueC[ABMP07], etc.). Some of them allow one to perform queries directly over the compressed representation of the text (either sequentially or using indexes), while others need to decompress the data (either fully or partially) before operating over them. However, despite the large amount of research developed along the years on this compression area, today there is an stated lack of available practical solutions [Sak09].

A more novel approach has been to combine compression and indexing, creating *self-indexed* representations of the text [NM07], in such a way that the compressed data represents at the same time the structured text and an index built over it. In recent works [FLMM05, FLMM06] a self-index for XML data was presented (XBzipIndex). This solution provides some query support, yet it is restricted to a very limited class of queries. In [ACM$^+$10], authors proposed another up-to-date proposal for compressed indexing of XML data. This tool, called SXSI, was tailored to work in main memory and it has been proved to be able to cope with an important subset of queries. This time, the main inconvenience is that its space requirements are still high compared to the size obtained by a plain compressor.

Hence, we can observe that efficient, scalable and stable implementations that take little space and provide, at the same time, full XML query support, are highly desirable, yet not satisfactorily achieved.

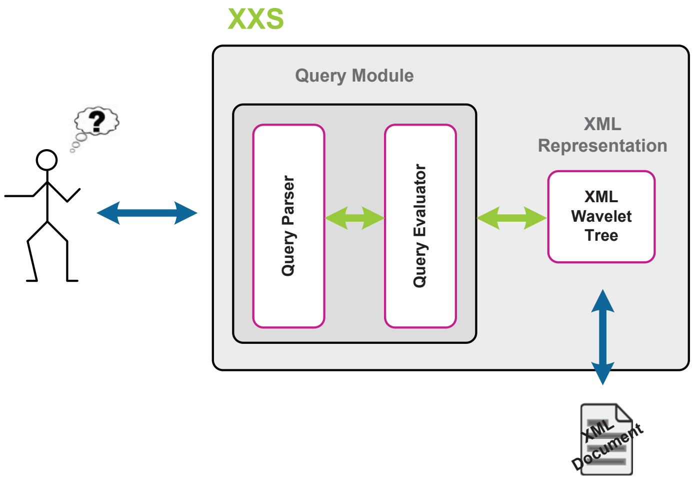


Figure 1.1: XXS architecture overview.

1.2 Contributions

This thesis addresses the open problem pointed out at the end of the previous section, and proposes a complete and competitive solution that efficiently supports XPath queries over a compressed and self-indexed representation of XML documents. We have developed a system, called XXS (***X****Path evaluation on* ***X****ML documents using a* ***S****elf-index*), that implements this solution. Figure 1.1 shows the architecture of our proposal. As it can be seen, it is mainly composed by two parts, which constitute the two main contributions of this work:

- *XML Representation*: The first contribution is a new data structure, we call XML Wavelet Tree (XWT), that provides compact representation of XML documents, with implicit self-indexing capabilities. Its construction is made in two phases. First, an initial pass on the input document[1] is performed to obtain the different words and frequencies, but keeping separated vocabularies depending on the category of the words, according to the different components of the XML data model. Then words are assigned a codeword using a

[1]Notice that a collection of documents can be regarded as a single document that integrates all of them.

variant of a word-based byte-oriented compressor, called the (s,c)-Dense Code [BFNP07], particularly tailored to make XWT suitable for querying purposes. The second pass replaces each word of the document by its corresponding codeword, yielding a compressed representation. Yet, the bytes of each codeword are not consecutively stored. Instead, they are placed along different nodes of a tree, following a WTBC codeword bytes reorganization [BFLN12].

XWT represents XML documents using only about 30%-40% of the original document size, which is a negligible overhead compared with the compression ratios achieved by the underlying compression method, as experiments prove. What is more striking is that XWT self-indexing properties and construction features lend this representation the ability to efficiently support XPath queries.

This new representation was published in preliminary form in the 13^{th} European Conference on Digital Libraries (ECDL 2009) [BCPN09].

- *Query Module*: As stated, XWT is a new approach to represent and process XML documents, in a time and space efficient way. But we have also addressed the query needs, by designing and implementing a query module for the efficient evaluation of XPath queries over the XWT representation. The *Query module* has two main components: the *Query parser* and the *Query evaluator* (see Figure 1.1). The *Query parser* submodule starts by obtaining a preliminary representation of the query, the *query parse tree*, that directly results from the own query syntax parsing. Then, several transformations are applied over this representation to produce another equivalent, but optimized one, that exploits XWT features, the *query execution tree*. This final representation constitutes the execution plan of the query.

  Once the *query execution tree* is obtained, the *Query evaluator* submodule directly translated it into operators that perform the global execution process over the XWT representation of the document. Three main strategies characterize the general evaluation procedure: a *bottom-up* approach, together with a *lazy evaluation* scheme, and the use of an *skipping* strategy. We describe in detail the whole process, and also the implementation of every operator.

The overall performance of XXS has been tested and compared with some well known state of the art solutions supporting XPath. Results show that it provides outstanding XPath evaluation capabilities, using little extra space (about 4%-8% of additional space) on top of the XWT representation.

A general description of the XXS system has been presented in the 7^{th} Workshop on Compression, Text, and Algorithms of the 19^{th} International Symposium on String Processing and Information Retrieval (SPIRE 2012).

1.3 Structure of the Thesis

After this introductory chapter, the rest of the thesis is organized in two main parts, as follows:

- **Part I - Basic Concepts and State of the Art Revision**: this part introduces some previous concepts for a better understanding of the rest of the thesis.

  - Chapter 2 introduces the basic concepts about XML documents. It presents a general overview of the *eXtensible Markup Language*, together with a brief description of the most important languages to process XML documents, from which the XPath query language is given an special focus.
  - Chapter 3 addresses the relevance and benefits of text compression nowadays to cope with space limitations, improving efficiency. Given the space challenge that may result from XML *verbosity*, compression becomes crucial. We present a revision of some basic notions about general text compression, and describe some classical and up-to-date proposals in this field.This chapter also explains some background information related to succinct data structures, and describes the most important ones regarding the scope of our work, namely, those used to solve basic operations (in particular, *rank* and *select* operations) over bit and byte sequences, as well as different succinct tree representations.
  - Chapter 4 aims to revise some of the most relevant proposals in the state of the art devoted to XML storage and querying. First a classification of some well-known *streaming* and *indexed* systems developed to specifically provide an efficient support for XML query languages is presented. Then the chapter also focuses on space aspects, and describe several works that have addressed the problem of minimizing space requirements, in the form of XML queriable and non-queriable compression techniques.

- **Part II - The XXS proposal**: this part is devoted to explain the contributions of this thesis that together constitute the core of the XXS system, and to experimentally evaluate its performance.

  - Remember that our proposal aims at providing compact representation of XML documents, with an efficient query support. As shown in Figure 1.1, two main parts compose it: the *XML representation* and the *Query module*. Chapter 5 focuses on the first one, and presents the XML Wavelet Tree (XWT), the compressed data structure we developed to represent XML documents with self-indexing capabilities. It describes in detail the XWT construction process, and the basic procedures to compress, decompress, and search words and phrases over that representation. This chapter also points out some of the XWT main properties that are key to further provide efficient query evaluation.

- The *Query module* of XXS is initially addressed in Chapter 6. In particular, this chapter deals with the *Query parser* component. The practical subset of XPath targeted in this work is first introduced, and then the process from query parsing to the production of the final query execution plan is described.
- Chapter 7 focuses on query evaluation, and closes our proposal with the description of the *Query evaluator*, the XXS submodule in charge of the efficient evaluation of XPath queries over an XWT representation. The chapter conceptually describes how the general evaluation procedure operates, combining a *bottom-up*, and *lazy evaluation* approach with a *skipping* strategy to avoid the processing of those parts of the document that are not relevant for a given query.
- Once known the description of the global execution process, Chapter 8 describes in detail every operator implementation, and their most relevant features.
- Chapter 9 benchmarks XXS, and analyzes both its compression properties, that stem from the underlying XWT representation, and its querying capabilities, by comparing it with some of the best current alternatives in the state of the art.

After that, this work ends with a final summary chapter and different appendixes, we next detail:

- Chapter 10 summarizes the main contributions of our work, and future directions of research.
- Appendix A lists the publications and other research activities related to this thesis.
- Appendix B describes the pseudocode of some of the operators presented in Chapter 7.
- Following the rules for PhD dissertation in a foreign language at the University of A Coruña, Appendix C contains a description, in Spanish, of this thesis work.

Part I

Basic Concepts and State of the Art Revision

Chapter 2

XML and XPath Query Language

This chapter presents the basic concepts related to XML documents. We first provide a complete overview of the *eXtensible Markup Language* in Section 2.1, by describing the main features of this specification. Then, Section 2.2 starts by introducing a brief description of some of the most important languages used to process XML documents, to next focus on the XPath query language. For this query language, its base data model (Section 2.2.1), as well as the syntax used to create XPath expressions are shown (Section 2.2.2). Finally, recent and further extensions to XPath are also presented in Section 2.2.3.

2.1 XML Overview

The *eXtensible Markup Language* (XML) is a World Wide Web Consortium (W3C) standard markup language that was originally defined as a simplified subset of the *Standard Generalized Markup Language* (SGML) for use on the World Wide Web. Since its first introduction in 1998 [XMLa, GP98], the language and its data model have soon proved their suitability to be the basis for the data interchange on the Internet. Today, XML is widely employed as a basic data model for representing general semi-structured information in different domains, ranging from business and e-commerce applications, to biology and chemistry areas.

The XML specification defines a set of rules for designing documents that can be processed by computer programs, while keeping human-readability. XML documents are basically built from strings of text and markups. The basic markup unit, which describes the structure of a document, is called an *element* (or *tag*). It is defined by a pair of matching marks, namely the *start-tag* and the *end-tag*, that enclose the element *content*. Start-tags begin with '<', and end-tags with '</'. Both are then followed by the name which identifies the element itself, and are closed by '>'. The name of the elements is generally related to the nature of the content they surround. For instance, we show below an example:

```
<section>XML Overview</section>
```

Some elements may be empty, that is, they have no content. In this case, we call them *empty elements*, and they are represented by combining the start-tag and the end-tag into a single *empty-element tag* beginning with '<', but ending with '/>'. There is also an special element, the so-called *root element*. It is the first element in the document and contains all the other elements of the document.

Elements can have *attributes*. They consist of *name-value* pairs, that appear within the *start-tag*, just after the name of the element. Names are separated from values, which are enclosed in single or double quotation marks, by '='. For example:

```
<section number="1">
  <title>XML Overview</title>
  <image file="document.png" caption="XML document sample"/>
</section>
```

Notice that `image` is an empty element, thus without content, but with two attributes, `file` and `caption`, whose values are *"document.png"* and *"XML document sample"*, respectively.

The elements and attributes names of some XML documents may be taken from multiple XML applications. Thus, they may share a common name, but standing for different meanings. In those cases, the use of *XML namespaces* allow one to disambiguate elements and attributes with the same name from each other by assigning them to URIs. Namespaces are implemented by attaching a *prefix* to each element and attribute name, which is mapped to a URI by using a `xmlns:`*`prefix`* attribute either in the elements in which they are used or in the XML root element. In the following example, the `xmlns:bk` attribute associates `bk` prefix to the URI "`http://www.vocexample.org/bkvoc`", and hence all element and attribute names prefixed by `bk` are in the same namespace:

```
<bk:catalog xmlns:bk="http://www.vocexample.org/bkvoc">
  <bk:journal>
    <bk:title>Information Retrieval</bk:title>
    <bk:year>2011</bk:year>
    <bk:citations>7024</bk:citations>
  </bk:journal>
</bk:catalog>
```

Some other important markups that can be found in an XML document are *comments* and *processing instructions*. Comments begin with '<!--' and end with '-->'. They may appear anywhere in a document outside of other markup. They are not part at all of the textual content of a document, since comments aim to make the raw XML more legible to human readers. XML processors may or may not retrieve the information included into the comments. Here is an example:

```
<library>
  <!--This content has been manually generated-->
  <book>
    <title>Three ways to capsize a boat</title>
  </book>
  ...
</library>
```

On the other hand, processing instructions (referred as PIs), that appear enclosed by '<?' and '?>', provide information to particular applications that may process the document. The application for which a processing instruction is intended, is identified immediately after the initial '<?', with a name called the *PI target*. The rest of the processing instruction contains the data with the instructions to be passed to the corresponding application. Like comments, processing instructions may appear anywhere in an XML document, outside of other markup. A common example of processing instruction is `xml-stylesheet`, which allows one to attach stylesheets to documents. For instance, in the following sample, the `xml-stylesheet` processing instruction indicates that browsers should apply the CSS stylesheet *book.css* to the document before showing it to the user:

```
<?xml-stylesheet href="book.css" type="text/css"?>
<library>
  <!--This content has been manually generated-->
  <book>
    <title>Three ways to capsize a boat</title>
  </book>
  ...
</library>
```

It is forbidden to start a processing instruction with the PI target `xml` (not either with `XML`, `XmL`, `xMl`, etc.), since this name is reserved to specify the *XML declaration* of an XML document. It constitutes the *prolog*[1] that any XML document should have[2], and provides information about the document itself:

[1]The prolog is everything in the XML document before the root element start-tag.

[2]An XML document does not have to have an XML declaration. Notwithstanding, if an XML document has it, then the declaration must be the first thing in the document.

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

When working with characters that are interpreted in a specific way, like the character '<', that is always recognized as the beginning of a start/end-tag, it is necessary to provide *escape* facilities to include them out of their actual scope. To this aim, XML defines the *entity references*, that allow escaping markup characters appearing within the text content or within attribute values. XML has five predefined entity references: *i*) `<`, to replace '<', *ii*) `&`, used instead of '&', *iii*) `>`, representing '>', *iv*) `"`, for ", and *v*) `'`, to substitute '. Only `<` and `&` must be used instead of the literal characters inside elements content. The others are optional. In turn, `"` and `'` are useful inside attribute values in order to avoid misconstruing the ending of the value.

An alternative to the use of entity references inside large blocks of text containing many occurrences of special characters are *CDATA sections*. A CDATA section begins with `<![CDATA[` and ends with `]]>`, and makes data to be processed simply as character data, but not as markups. That is, markups are ignored. For instance, let us consider an XML document including some samples of source code. They may contain characters that an XML processor would recognize as markups (e.g. & and '<'). We can use a CDATA section to enclose the samples, and to prevent the usual performance:

```
<![CDATA[
  a = i << 3;
  *b = &a;
]]>
```

2.1.1 Well-formedness and Validation

As stated, XML specifies a set of rules that make up the grammar of an XML document. Besides the possible components, it determines for instance, where elements may be placed, which names are allowed, how attributes are included, and so on. Documents that fulfill the grammar are said to be *well-formed*. There are many rules, but some of the most important ones that a *well-formed* XML document must satisfy are the following: *i*) it has an *unique* root element, *ii*) every start-tag has its matching end-tag, *iii*) elements can not overlap (i.e. an element can not be closed until all the elements it contains have been closed), *iv*) attribute values must be quoted, *v*) an element may not have two attributes with the same name, *vi*) markup characters '<' and '&' may not occur in the character data of elements and attributes. Notice that the three first rules induce a proper tree structure on an XML document. Figure 2.1 illustrates an example. Furthermore, the grammar sets the basis needed to create XML parsers, able to read any XML document.

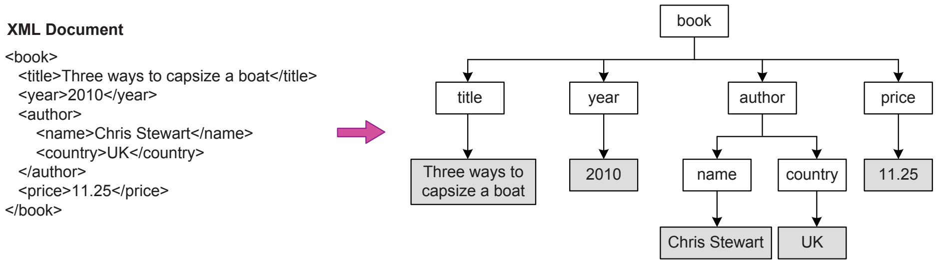


Figure 2.1: Tree view of a sample XML document.

There are, basically, two main APIs for XML. The *Simple API for XML* (SAX) [SAXa] is an event-based API. It sequentially scans an XML document and throws events that are further handled by the parser. Examples of events are, for instance, an occurrence of a start-tag or an end-tag, content characters, a processing instruction, a comment, etc. In contrast, the *Document Object Model* (DOM) [DOM], is another API that builds a tree representation of the entire document in memory, thus using much more memory than the former approach, but permitting to randomly access and manipulate the document.

In addition to being well-formed, an XML document may also be *valid*. Particular XML applications may need to ensure that a given XML document adheres to some guidelines (rules) imposed by the application itself. In that case, the allowed markups, as well as their composition are specified in a *schema*. Whenever an XML document matches the schema it is said to be *valid*. If not, we say that the XML document is *invalid*. Hence, the *validity* of a document depends on which schema is used to compare it with. Documents do not always need to be valid, for many applications it is enough that the document is well-formed. There are several XML schema languages, each one having different levels of expressiveness. The most widely supported XML schema language[3] is the *Document Type Definition* (DTD). A DTD defines the list of markups (e.g. elements, attributes, entities, etc.) that can be used in a document, and how they can be combined, together with basic content specifications. For example:

```
<!ELEMENT library   (book+)>
<!ELEMENT book      (title, summary, chapter*)>
```

[3] It is also the only one defined by the XML specification [XMLa].

```
<!ELEMENT title     (#PCDATA)>
<!ELEMENT summary   (#PCDATA | keyword)*>
<!ELEMENT chapter   (#PCDATA)>
<!ATTLIST book       ref   CDATA #REQUIRED
                     href  CDATA #IMPLIED>
```

The first element declaration of the DTD sample above states that each `library` element must contain one or more `book` child elements[4]. In turn, the second line indicates that each `book` element must have exactly one `title` child element followed also by exactly one `summary` element, and zero or more `chapter` elements[5]. That is, every `book` must contain a `title` and a `summary`, and may or may not have a `chapter` or multiple `chapter` elements. Nevertheless, the `title` must come before the `summary`, and this one must appear before all chapters.

Regarding `title` and `chapter` elements, lines 3 and 5 say that each occurrence of any of these elements may only contain *parsed character data* (referred with `#PCDATA`), that is, raw text, but not any child element. In case *mixed content* is allowed, then we use an element declaration similar to that shown in line 4. This states that a `summary` element may contain parsed character data as well as `keyword` children. It does not specify in which order they appear, nor how many instances of each occur. This declaration allows a `summary` to have 0 `keyword` children, 1 `keyword` children, or 26 `keyword` children.

In addition, the use of ATTLIST declarations are used to declare element attributes. For instance, if we consider lines 6 and 7 of the sample DTD we have been analyzing, they indicate that any `book` element must have a `ref` attribute (`#REQUIRED`). However, the `href` attribute is optional (`#IMPLIED`), and may be omitted from particular `book` elements. Both attributes are asserted to contain character data (i.e. any string of text) [6].

Therefore, according to the DTD sample just seen, the following XML document would be valid:

```
<library>
  <book ref="CHS001">
    <title>Three ways to capsize a boat</title>
    <summary>A charming and lyrical read, awash with the joy
    of discovery</summary>
    <chapter>The proposal</chapter>
    <chapter>When dreams come true</chapter>
    <chapter>Sailing to Greek Islands</chapter>
```

[4] The '+' after `book` stands for "one or more".

[5] This time '*' after `chapter` denotes "zero or more".

[6] `CDATA` is the most generic attribute type. Other attribute types are: `NMTOKEN`, `NMTOKENS`, `Enumeration`, `ENTITY`, `ID`, `IDREF`, etc.

```
    ...
  </book>
</library>
```

However, it would not be the case of the next document, since the `summary` element comes before the `title` one, and also the `book` element does not have the mandatory attribute `ref`:

```
<library>
  <book>
    <summary>A charming and lyrical read, awash with the joy
    of discovery</summary>
    <title>Three ways to capsize a boat</title>
    <chapter>The proposal</chapter>
    <chapter>When dreams come true</chapter>
    <chapter>Sailing to Greek Islands</chapter>
    ...
  </book>
</library>
```

Usually schemas are supplied in separated files from the documents they describe. Yet, DTDs are the only ones that can also be included inside the XML document. In both cases, the XML markup corresponding to the *document type declaration* is used. It is included in the prolog of the XML document, just after the XML declaration and before the root element, and it allows one to specify either a reference to an external DTD to which the document should be compared or even the DTD itself (between square brackets). For instance, let us assume that the previously discussed sample DTD is available at `http://dtdsamples.com/library.dtd`. Then, the document type declaration of an XML document conforming to this DTD looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?xml-stylesheet href="book.css" type="text/css"?>
<!DOCTYPE library SYSTEM "http://dtdsamples.com/library.dtd">
<library>
  ...
</library>
```

This document type declaration tells that the root element of the document is `library` and that the DTD for the document can be found at `http://dtdsamples.com/library.dtd`.

Nevertheless, DTDs may not always be enough, since they provide limited support for type definition of the contained data. That is, a DTD does not allow one to specify, for instance, that an element contains a real number or a date range. Some other well-known and more powerful schema languages that permit these kind of constraints are the *W3C XML Schema Language* [XSD], *RELAX NG* [CM01] or *Schematron* [Sch].

2.2 XPath Query Language

There are several languages for processing XML documents: the *XML Path Language* (XPath) [XPaa], the *XML Query Language* (XQuery) [XQu], the *XSL Transformation* (XSLT) [XSL], the *XML Linking Language* (XLink) [XLi], the *XML Pointer Language* (XPointer) [XPo], etc. The reference query languages are both XPath and XQuery[7], while XSLT is used to transform an XML document into another XML document, by means of template rules. In turn, XLink allows one to attach simple, bidirectional or even multidirectional links to XML documents, with can be further specified by using XPointer, that permits to address individual parts of an XML document. As it can be seen, each one deals with different aspects of XML processing, yet the relevance of XPath stems from the fact that it constitutes the base for most of the rest ones. Since this thesis is focused on this language, we will next describe it in detail[8].

2.2.1 XPath Data Model

XPath aims to select parts of XML documents. The *XPath data model* considers XML documents as trees made up of nodes of different types. There are basically seven node types: *i*) the *root node*, *ii*) *element nodes*, *iii*) *text nodes*, *iv*) *attribute nodes*, *v*) *comment nodes*, *vi*), *processing instructions nodes*, and *vi*) *namespace nodes*. There is always a *root node* which is the root of the hierarchy. It has no name and no parent, and its unique child is the *element node* representing the *root element* of the document. It may also contain any comment or processing instruction occurring before the root element start-tag or after the root element end-tag.

Element nodes represent the elements of an XML document. Each of them has a parent, which in case of the root element is the *root node*, and for the rest of the element nodes, is the node containing it. An element node may have children

[7] The main expression of XQuery is the FLWOR expression: FOR, LET, WHERE, ORDER, RETURN. This expression supports iteration and binding of variables to intermediate results. It is commonly assumed that the FLWOR expression serves approximately the same purpose in XQuery than the SELECT expression serves in the SQL language for relational databases.

[8] According to XPath 1.0 [XPaa].

that can be nodes representing another elements, text, comments, and processing instructions directly contained by the element.

Each attribute makes up the corresponding attribute node. The parent of an attribute node is the element node it belongs to, still an attribute is not considered its child. Attribute nodes have no children. The textual content of an element is represented by text nodes. Each text node contains the maximum contiguous run of character data not interrupted by any tag. Like the attribute nodes, text nodes do not have child nodes. Finally, each of the comment nodes, processing instruction nodes and namespace nodes, are related to occurrences of the respective components their names refer. Yet, these are rarely handled.

2.2.2 XPath Expressions

The basic concept in XPath is the *expression*. XPath syntax mainly consists of expressions whose result is usually a set of nodes[9], but it can also be a boolean, numeric or string value. That is, expressions allow one to specify a set of nodes and optionally a function on the result. Hence, it is possible to search, for instance, for all the `book` nodes in an XML document and just deliver the set, or add a counting operation and deliver instead the number of such nodes.

The most important XPath expression is the so-called *path expression*, also known as *location path*. A location path identifies a set of nodes in a document and is composed by a sequence of one or more minor units, namely the *location steps*. Location paths may start by a slash, '`/`', in which case they are *absolute location paths*, that are evaluated from the document root node, or may be *relative location paths*, which are evaluated from a context node.

Before formally describing path expressions, let us consider the following example of location path related to the XML document of the Figure 2.2 to show how they work:

```
/store/city/books[./@category="fantasy"]/book/title
```

Since the expression begins with a slash, its evaluation will start from the root node. In particular, we are interested in `store` element nodes that are children of the root node. In that case, the element `store` (line 1) of the sample XML document satisfies this constraint, so we select it. Then, the following step selects all its children of type `city` (lines 2 and 23) and, for each selected node, the next step obtains those elements nodes that are `books` child nodes. However, the expression

[9]According to XPath 1.0 [XPaa] results are node sets, hence with no order; while in XPath 2.0 [XPab], results are *sequences* of nodes in a particular order, the 'document order' (which applied over the XML document structure corresponds to a preorder traversal). However, arguably all the systems supporting XPath 1.0 assume as well this 'document order' for results delivering. In this work we also assume that, even as a way to allow the compatibility of our system with future extensions.

XML Document

```
1.  <store>
2.    <city name="Coruña"  province="Coruña">
3.      <books category="fantasy">
4.        <book year="1997">
5.          <title>Harry Potter and the Philosopher's Stone</title>
6.          <author>J.K. Rowling</author>
7.          <price>10.95</price>
8.        </book>
9.        <book year="2000">
10.         <title>Harry Potter and the Goblet of Fire</title>
11.         <author>J.K. Rowling</author>
12.         <price>13.50</price>
13.       </book>
14.     </books>
15.     <books category="literature">
16.       <book year="1999">
17.         <title>Driving over Lemons: An Optimist in Andalucia</title>
18.         <author>Chris Stewart</author>
19.         <price>10.25</price>
20.       </book>
21.     </books>
22.   </city>
23.   <city name="Vigo"  province="Pontevedra">
24.     <books category="fantasy">
25.       <book year="1954">
26.         <title>The Two Towers</title>
27.         <author>J.R.R. Tolkien</author>
28.         <price>20.15</price>
29.       </book>
30.       <book year="1955">
31.         <title>The Return of the King</title>
32.         <author>J.R.R. Tolkien</author>
33.         <price>23.75</price>
34.       </book>
35.     </books>
36.   </city>
37.   <city name="Santiago"  province="Coruña">
38.   </city>
39. </store>
```

Figure 2.2: Example of XML document.

surrounded by the square brackets restricts that selection to only those `books` element nodes that have an attribute `category` with value *fantasy* (lines 3 and 24). Once those elements are retained, we continue by selecting their `book` children (lines 4, 9, 25 and 30). At last, the final step returns the `title` child nodes of each of them (lines 5, 10, 26 and 31).

Now we will discuss in detail the main features of the path expressions. As seen, successive location steps are separated by slashes, and are evaluated from left to right. Each step in the path is relative to the one that preceded it. That is, the result of each location step makes up the *context* for the next. The general pattern of a location step is given by $/axis::node\_test[predicate]$. That is, it is composed of three main parts:

- An *axis*, that specifies how to move from the context node to look for new nodes. There are 13 different axes, from which the 8 most common are illustrated in Figure 2.3:

  1. `child`: identifies every child node of the context node[10].
  2. `descendant`: selects every child node of the context node, their children, and so on. That is, this axis identifies every descendant node of the context node[10].
  3. `parent`: the parent node of the context node.
  4. `ancestor`: identifies the parent node of the context node, but also the parent of the parent node, and so forth until reaching the root node.
  5. `following`: selects every node[10] that appears, in document order, *after* the context node, excluding all its descendant nodes.
  6. `preceding`: identifies every node[10] that appears, in document order, *before* the context node, excluding all its ancestor nodes.
  7. `following-sibling`: every node[10] sibling of the context node that appears *after* the context node, in document order.
  8. `preceding-sibling`: identifies all nodes[10] siblings of the context node appearing *before* the context node, in document order.
  9. `attribute`: selects every attribute node of the context node. This axis can only be applied to element nodes.
  10. `self`: identifies the context node itself.
  11. `descendant-or-self`: identifies the context node and all its descendants.
  12. `ancestor-or-self`: selects the context node and all its ancestors.
  13. `namespace`: identifies all the namespace nodes belonging to the context node. The context nodes can only be element nodes.

  Usually axes are classified into *forward axes* and *reverse axes*, depending on whether they take nodes that, in document order, are *after* or *before* the context node, respectively. Thus the `child`, `descendant`, `descendant-or-self`,

[10] Other than attributes and namespace nodes.

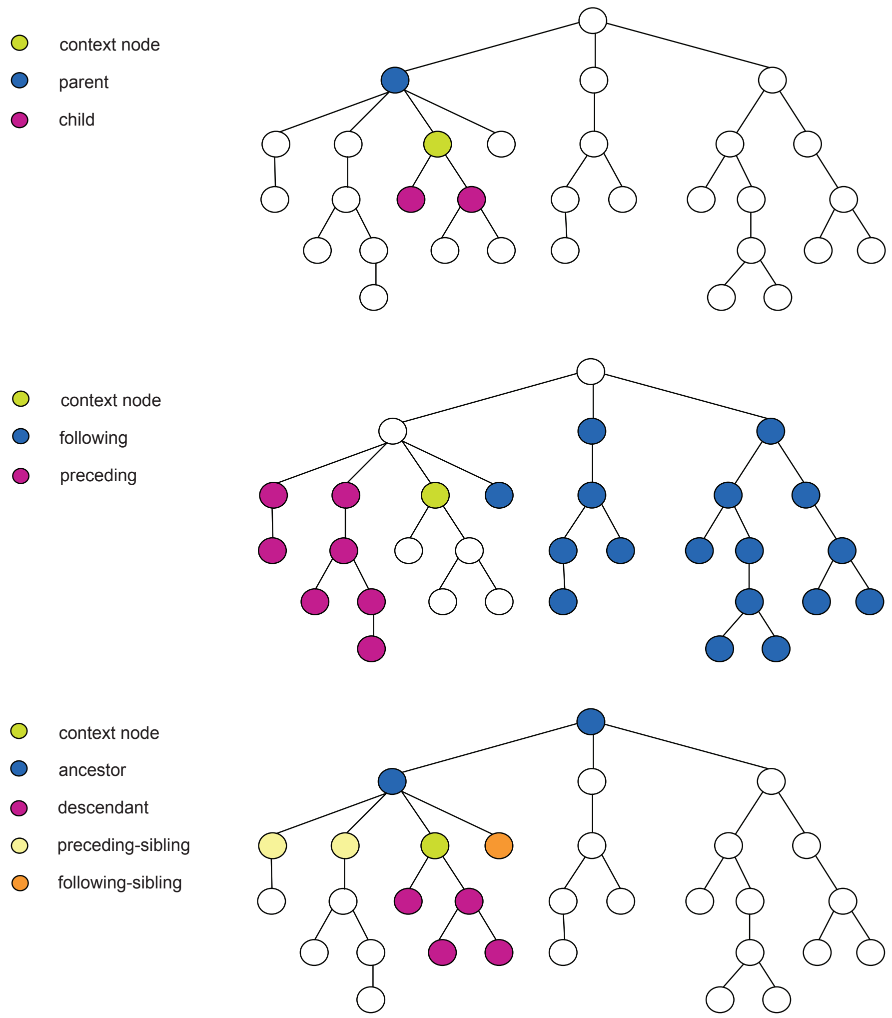


Figure 2.3: Examples of XPath axes.

`following`, `following-sibling`, `attribute` and `namespace` axes, are all forward axes. In turn, `parent`, `ancestor`, `ancestor-or-self`, `preceding`, and `preceding-sibling`, are all considered as reverse axes. Note that the `self` axis can be either classified into the forward axes category or into the reverse axes group.

Some of the axes admit an abbreviated form. For instance, whenever the axis is omitted after '/', as happened in the example previously shown (e.g. `/store/city/books[./@category="fantasy"]/book/title`), a `child` axis is assumed (since it is by far the most commonly used). `Attribute` axis, can also be expressed by the symbol `@`. Likewise, `self` and `parent` axes are represented with a shorter notation by using a single period ('`.`'), and a double period ('`..`'), respectively.

- A *node test*, that indicates the name or the type of the nodes that should be selected along the axis.

  Every axis has a *principal node type*. If an axis can contain elements, then the principal node type is element; otherwise, it is the type of the nodes that the axis can contain. That is, for the `attribute` axis the principal node type is attribute, for the `namespace` axis, it is namespace, and for the rest of the axes, the principal node type is element. Commonly, node tests specify the name that the selected nodes must have. In this scenario, the name test is fulfilled if the type of the node corresponds to the principal node type of the axis specified in the location step and if its name matches that of the test. For example, let us assume the location step marked in blue face in the following path expression: `/store/descendant::book`. Then, according to it, only `book` element nodes descending from a `store` element node are selected.

  It is also possible to use the wildcard symbol '`*`', instead of a specific name. In such a case, the name test is true for any node of the *principal node type*, no matter its name. For instance, if we consider the example of Figure 2.2, the last location step of the path expression `/store/city/books/book/*` will select `title`, `author` and `price` element nodes[11] children of any `book` node fulfilling the conditions imposed by the rest of the previous location steps. Likewise, `/store/city/@*`, will select any attribute node, regardless its name, from a `city` element node child of `store`. Assuming again the example of Figure 2.2, both `name` and `province` attributes of the corresponding cities will be delivered by this path expression.

  In addition, node type tests allow selecting nodes of a specific type. Different functions are used to represent the node types we are interested in. For example, `node()` stands for nodes of any type, and `text()` selects only text nodes, while `comment()` and `processing-instruction()` are used to select comment nodes and processing-instruction nodes, respectively.

  As stated in previous examples, when axes are specified through their shorthand notations, the axis and the node test are combined in the location step. For instance, that is the case of `/store/city/@name`. However, if

[11] Note that these three different element nodes correspond to all the types of element nodes child from a `book` element in the document sample.

the unabbreviated form is used, two colons `'::'` separate the axis from the node test. If we consider the same example, and use the unabbreviated syntax, we will rewrite it as `/child::store/child::city/attribute::name`. Another typical abbreviation is the double slash, `'//'`, which stands for `'/descendant-or-self::node() /child::'`.

- Zero or more *predicates* (also called *filters*) used to further refine the node selection. Predicates are enclosed in square brackets, and may contain any XPath expression, from whose result a boolean value is determined[12]. The predicate is tested for each node of the current node set, selected at the step to which the predicate is applied. If the predicate is evaluated to true, then the node is kept in the current set. Otherwise, it is discarded. For example, `//book[./@year="2000"]`, will deliver only those books whose publication year is equal to 2000 (line 9). Another example could be `//book[./price]`, which returns only those books for which a `price` child node exists (in this XML document sample, all books have a price, hence they are all delivered). We have just seen the use of the equal sign inside a predicate, `'='`. Yet, XPath supports other relational operators such as `'<'`, `'>'`, `'!='`, etc. In addition, predicates can be logically combined by using `'and'`/`'or'` operators. Let us consider the path expression `//city[./@name="Coruña" or ./@name="Santiago"]//book/title`. It will select the titles of books that can be bought in the company stores placed in Coruña and Santiago (since there are still no books in Santiago store, this query only delivers the titles of those books we can find in Coruña, that is, lines 5, 10 and 17).

Moreover, XPath provides a number of built-in functions that can be used as part of an step expression. The most common functions are those that operate on node sets, such as `count()` or `position()`, and those representing basic string operations, like `contains()` and `starts-with()`. Still, boolean and number functions are also supplied. An example of use of these functions is shown by the following path expression: `count(/store/city[./@name="Coruña"] /books/book[contains(./title, "Potter")])`, that delivers the number of books that can be acquired in stores of Coruña and whose title contains the word `'Potter'`. Thus, applied to the XML document sample shown in Figure 2.2, the answer will be 2.

[12] If the type of the expression result is other than boolean, different situations arise. For instance, if it is a number, then the predicate is true if and only if the value of the number matches the context position. If the type of the expression result is a string, then the predicate is evaluated to true if and only if the length of the string is greater than zero. Finally, if the type of the expression result is a node set, then the predicate is true if and only if the node set contains at least one node.

2.2.3 XPath Extensions

Recently, XPath has been enriched with full-text search capabilities [Ful], that allow one to perform text-based searches considering some special operators. Although they are not addressed in this thesis, we briefly discuss the four different categories in which these operators are roughly divided:

- *Word expansions*: to search for a particular word/term, but also for other terms related to the query term. That is the case of applying a *stem* operation or of searching for close terms in a *thesaurus*.
- *Matching options*: to define the "factors" that stand for a specific kind of match. For instance, to include a *case* option, that indicates how uppercase and lowercase characters are considered, or to introduce *wildcards*.
- *Positional operations*: to search for occurrences of query terms that are "near". This proximity can be specified by providing a scope (e.g. within the same *sentence* or the same *paragraph*), the exact distance between terms, a distance range, and even the order of the terms to match.
- *Combining operations*: to support logical combinations, namely *and*, *or*, *not*, and *not in*, of full-text selections.

Chapter 3

Text Compression and Succinct Data Structures

Data compression constitutes a key factor for the efficient handling of increasing amounts of available information. Not only does it allow saving storage space, but also time. This fact is even stressed in case of XML documents, which due to their verbosity may result into huge size documents, that have to be transmitted, stored, and also queried. Most of the XML compression works are based on general text compression techniques. Hence, the first part of this chapter starts by providing a complete overview of basic text compression notions that are needed for a better understanding of forthcoming concepts and proposals described in the rest of this thesis. A brief description of several concepts related to Information Theory are first shown in Section 3.1.1 and Section 3.1.2. Then, Section 3.1.3 presents a taxonomy of the text compression techniques, of which the main ones are discussed in depth in Sections 3.1.4, 3.1.5, and 3.1.6. Finally, some measure units that can be used to compare compression techniques are also introduced in Section 3.1.7.

The second part of this chapter (Section 3.2) is devoted to the description of succinct data structures that aim to reduce space requirements, while keeping an efficient processing of the data. Section 3.2.1 describes some succinct data structures to solve basic operations (namely, *rank* and *select*) over bit and byte sequences, which are commonly used to improve the efficiency of other high-level structures, such as, for instance, the structures used to represent trees. These succinct tree representations are precisely further discussed in Section 3.2.2, given their relevance within XML contexts.

3.1 Text Compression

This section introduces some basic concepts about general text compression needed to better understand further explanations.

3.1.1 Concepts of Compression

The objective of *text compression* is to transform a source text into a representation containing the same information but whose length is as small as possible. To this aim, the source text is seen as a sequence of small fragments, called *source symbols* (e.g. characters, words, q-grams etc.), which are the basic units to compress. The amount of all different source symbols that appear in the text is known as the *source alphabet*, β.

A compression technique replaces each source symbol of the text by a *codeword*. Then the compressed text is the sequence of codewords assigned to its source symbols. The mapping between source symbols and codewords is given by an *encoding scheme* or *code*, that defines how each source symbol is *encoded*. Each codeword is composed by one or more *target symbols* from a *target alphabet*, Γ, of size D. Depending on D, the number of bits, b, needed to represent a target symbol, is different. For instance, codewords which are sequences of bits, that is, *bit-oriented* codewords, use $b = 1$ bits to represent each of the $D = 2^1$ symbols of Γ. Instead, *byte-oriented* codewords, which are sequences of bytes, need $b = 8$ bits, thus permitting to represent $D = 2^8$ distinct target symbols.

The process of restoring the source symbol corresponding to a given codeword is called *decoding*. When dealing with text, it is mandatory for the *decoding* algorithm to obtain an exact replica of the original source after decompression. In such a case, we refer to those methods as *lossless compression techniques*. However, there are some situations where the use of *lossy compression techniques* is allowed. For instance, image and sound compression are common examples of this kind of scenarios, since human visual/auditive sensibility cannot detect small differences between both the original and the decompressed data.

A *code* is said to be a *distinct code* if each codeword is distinguishable from every other, that is, the mapping from source symbols to codewords is one-to-one. A code is *uniquely decodable* if every codeword is identifiable from a sequence of codewords. For instance, let us assume a source alphabet, $\beta = \{a, b, c, d\}$, and the following encoding scheme: $a \leftrightarrow 0, b \leftrightarrow 1, c \leftrightarrow 10, d \leftrightarrow 11$. It is a distinct code, however it is not uniquely decodable, since the sequence 110 could be decoded either as $1, 1, 0$ (bba), $1, 10$ (bc) or $11, 0$ (da). In turn, if we consider the same source alphabet, but a different code: $a \leftrightarrow 00, b \leftrightarrow 10, c \leftrightarrow 01, d \leftrightarrow 011$, we will note that this new encoding scheme fulfills the uniqueness condition. However, it is still required to perform a *lookahead* during decoding to observe that, for instance, the sequence 01100001 corresponds to *cbac*, since we can not determine that the first codeword is 01 (c), and not 011 (d), up to analyzing some of the binary symbols beyond the codeword itself[1]. A uniquely decodable code is called a *prefix code* (or prefix-free code) if there is no codeword being a proper prefix of any other codeword. Assuming again the source alphabet $\beta = \{a, b, c, d\}$, the mapping $a \leftrightarrow 00, b \leftrightarrow 10, c \leftrightarrow 110, d \leftrightarrow 111$, is an example of prefix code. An important property of prefix codes is that they are *instantaneously decodable*. That is, an encoded message can be parsed into codewords without the need for lookahead, thus permitting decoding a codeword right after it is read, which improves decoding speed. For example, a binary string like 00101101000 is univocally and instantaneously decoded to *abcba*, using the aforesaid prefix code, with no inspection of the following code symbols to decode a codeword.

A prefix code is a *minimal prefix code* if, being x a proper prefix of some codeword, then $x\tau$ is either a codeword or a proper prefix of a codeword, for each target symbol τ in the target alphabet Γ. For instance, we have seen that the mapping $a \leftrightarrow 00, b \leftrightarrow 10, c \leftrightarrow 110, d \leftrightarrow 111$, is a prefix code. However, it is not minimal. Notice that since 0 is a proper prefix of 00, it will require 01 be either a codeword or a proper prefix of a codeword, but it is neither. If the codeword 00 is replaced by 0, then the code becomes a *minimal prefix code*. The minimality property prevents the use of codewords that are longer than necessary.

In association with prefix codes, *Kraft inequality* [Kra49] establishes whether it is feasible or not to find a prefix code with some codeword lengths. Let us denote l_{c_i} the length of a codeword c_i, then a binary prefix code with codewords $c_1, c_2, \ldots, c_n$, and with corresponding codeword lengths $l_{c_1}, l_{c_2}, \ldots, l_{c_n}$ exists if and only if $\sum_{i=1}^{n} 2^{-l_{c_i}} \leq 1$. The codeword lengths of a prefix code satisfy Kraft's inequality. Conversely, given codeword lengths $l_{c_1}, l_{c_2}, \ldots, l_{c_n}$ that fulfill Kraft's inequality, then a prefix code with those codeword lengths exists. Yet, any code satisfying Kraft's inequality does not have to be a prefix code. Let us consider the *uniquely decodable* encoding scheme $a \leftrightarrow 00, b \leftrightarrow 10, c \leftrightarrow 01, d \leftrightarrow 011$, previously characterized. The associated codeword lengths are $2, 2, 2, 3$, and hence they satisfy Kraft's inequality, since $2^{-2} + 2^{-2} + 2^{-2} + 2^{-3} = \frac{7}{8} \leq 1$. However it is not a prefix code. Notice as well that in case $\sum_{i=1}^{n} 2^{-l_{c_i}} = 1$, the codeword lengths are minimal, thus yielding to the existence of a minimal prefix code. For instance, if we consider the above shown *minimal prefix code*, $a \leftrightarrow 0, b \leftrightarrow 10, c \leftrightarrow 110, d \leftrightarrow 111$, we can check that $2^{-1} + 2^{-2} + 2^{-3} + 2^{-3} = 1$.

3.1.2 Entropy and Redundancy

As previously stated, compression techniques aim at representing the data by using less space [BCW90]. For that purpose, they try to exploit *redundancies* in the source text, while keeping the source information. The information included in a

[1] Notice that if the binary string of zeros after the third position had been of odd length, then the first codeword would have been 011 (d), and hence the unique valid input sequence *daac*.

source message is equivalent to the amount of *surprise* in the message. Shannon's work [SW49] established the basis of information measurement and transmission. Given a source symbol s_i, the amount of information associated is defined by $I(s_i) = -\log_D p(s_i)$, where $p(s_i)$, denotes the probability of occurrence of the symbol s_i, and D is the number of symbols of the target alphabet. The intuition is that the less likely the occurrence of s_i, the more *surprised* we are to observe it. For instance, if $p(s_i) = 1$, no information is obtained from that observation, since it is the expected outcome. Instead, if $p(s_i)$ tends to 0, we will be surprised by the occurrence of s_i, since it is a symbol which does not usually appear. Consequently, its observation has high information content.

If we are interested in quantifying the *expected amount of surprise* of a source alphabet, we can obtain it by computing its *entropy*, H. It is defined as $H = -\sum_{i=1}^{n} p(s_i) \log_D p(s_i)$, and provides the *average information content* of the source. That is, entropy indicates a lower bound on the number of target symbols per source symbol needed to encode a message[2]. Closely related to entropy, is the *redundancy*. Let us consider $l(c_i)$, the length of the codeword c_i assigned to the source symbol s_i, then *redundancy* is described as:

$$R = \sum_{i=1}^{n} p(s_i) l(c_i) - H = \sum_{i=1}^{n} p(s_i) l(c_i) - \sum_{i=1}^{n} -p(s_i) \log_D p(s_i)$$

In other words, redundancy is a measure of the difference between the *average codeword length* and the actual average information content (i.e. the *entropy*). Remember that compression techniques try to reduce the redundancy of the source messages. Since the entropy is determined by the distribution of probabilities of the source alphabet, the smaller the average codeword length of a code, the better the code is. A code having the minimum average codeword length is called a *minimum redundancy code*.

3.1.2.1 Entropy in Context-dependent Messages

In Section 3.1.2 we assumed that the source symbol probabilities $p(s_i)$ did not depend on the previously appeared symbols. That is, we assumed independence of source symbols and their occurrences. Yet it is possible to model the probability of a source symbol s_i in a more precise way, by considering the source symbols appeared before it. We call *context* of a source symbol s_i to a fixed-length sequence of source symbols preceding s_i. Depending on the length m of the considered *context*, different m-order models are defined, yielding also different kth-order (H_k) *entropy* expressions:

[2]Usually $D = 2$, hence the entropy gives us the minimum number of bits per symbol that will be required to encode a source text.

- *Base-order models* consider that all source symbols are independent and equally like to occur. Hence, the entropy for this scenario, denoted as H_{-1}, results $H_{-1} = \log_2 n$.
- *Zero-order models* assume that source symbols are still independent, but with frequencies given by their number of occurrences. In this case, the zero-order entropy is defined as $H_0 = -\sum_{i=1}^{n} p(s_i) \log_D p(s_i)$.
- *First-order models* compute the probability of occurrence of a source symbol s_j conditioned by the previous occurrence of the symbol s_i (that is, $P_{s_j|s_i}$). Then the arising *entropy* is obtained as $H_1 = -\sum_{i=1}^{n} p(s_i) \sum_{j=1}^{n} P_{s_j|s_i} \log_D(P_{s_j|s_i})$.
- *Second-order models* obtain the probability of occurrence of a source symbol s_k conditioned by the previous occurrence of the sequence $s_i s_j$ (that is, $P_{s_k|s_j s_i}$). Hence, for these models, the *entropy* is computed as $H_2 = -\sum_{i=1}^{n} p(s_i) \sum_{j=1}^{n} P_{s_j|s_i} \sum_{k=1}^{n} \log_D(P_{s_k|s_j,s_i})$.
- *Higher-order models* work in a similar way.

Some techniques combine distinct m-order models to estimate the probability of the next source symbol. *Prediction by Partial Machine* (PPM) [CW84, BCW90, Mof90], is an example of that kind of compressor which combines several finite-context models of order 0 to m.

3.1.3 Classification of Text Compression Techniques

Prior to establish a classification of the text compression techniques, it is important to separate the two main phases that compose the global compression process itself.

- *Modeling* : the source text is partitioned into symbols and their probability distribution is estimated, in order to try to discover something about the structure of the input. The more accurate the estimations of the probabilities are (e.g. by considering the *context* of symbols, as discussed in Section 3.1.2.1), the better the compression is. In the field of natural language text compression, the partition of the input into symbols can be done by considering either characters or words[3] as the basic units. Although a character oriented approach had been traditionally applied, obtaining poor compression ratios (around 65%), the benefit of using word-based models[4], to improve the compression achieved (around 25%-35%), was later shown [Mof89, TM97, dMNZBY00]. Two empirical laws characterize this performance:

[3]Also *q-grams* may be considered.

[4]One of the most popular is the *spaceless word model* [MNZBY98], that creates the vocabulary by considering the following premise: if a word is followed by a space we just encode the word, otherwise both the word and the separator are encoded.

- *Heap's law* [Hea78] provides an approximation of how a vocabulary grows as the size of the text collection increases. In particular, it settles that the relationship between the number of words in a natural language text (N) and the number of differen words (V) in that text (that is, words in the vocabulary), is defined as $V = kN^{\beta}$, where k and β are free parameters empirically determined. For example, in English text corpora, it usually holds that $10 \leq k \leq 100$ and $0.4 \leq \beta \leq 0.6$. Therefore, Heaps' law suggests that the price of having a larger set of source symbols (as it happens when using words instead of characters), is not significant on large text collections, as the vocabulary grows sublinearly.
- *Zipf's law* [Zip49] gives an estimation of the word frequency distribution for a natural language text. The frequency of the $i-th$ most frequent word in the vocabulary is given by the expression $f = \frac{t}{i^{\theta}}$, being $t = \frac{1}{\sum_{i>0} \frac{1}{i^{\theta}}} = \frac{1}{\zeta(\theta)}$ a normalization factor[5], and θ a constant that depends on the analyzed text ($1 < \theta < 2$). Then, following Zipf's law the distribution of words in natural language text is more skewed than that of characters [BYRN99].

Indeed, since IR systems are built taking the words as the basic elements, word-based compression techniques meet their requirements, and hence can be perfectly integrated with them.

- *Coding* : the encoding scheme assigns a codeword to each source symbol according to the probability biases obtained in the modeling phase.

According to this, text compression techniques can be categorized depending on the model used, but also on how the encoding process take place. Regarding the first criterion, compression techniques are classified as using:

- *Static or non-adaptive models* : source symbols are assigned fixed frequencies, taken from previously computed probability tables. These probabilities are usually drawn from experience, and do not fit the actual distribution of the source symbols in the input message, thus the encoding process yields, in general, poor compression ratios. Yet those models can be used in specific scenarios. The *Morse* code is a well-known example of this approach.

- *Semi-static models* : these models are commonly associated with *two-pass* techniques. The first pass over the text is performed to gather the different source symbols that compose the source alphabet or *vocabulary*, and to compute their frequency distribution. These probabilities are then used by the *encoding scheme* to create and assign a codewords to each source symbol.

[5] $\zeta(x) = \sum_{i>0} \frac{1}{i^x}$ is known as the *Zeta* function.

After that, a second pass takes place. The whole text is processed again, and source symbols are replaced by the corresponding codewords, leading to the compressed text, which is stored together with a header containing the mapping between symbols and codewords, needed for decompression. As it can be noticed, semi-static techniques are not able to compress streams of text, since the encoding can not start before the whole first pass has been completed. Some representative semi-static compression techniques are the classical Huffman-based codes [Huf52], and those based on Dense Codes [BFNP07].

- *Dynamic or adaptive models* : usually known as *one-pass* techniques, these methods do not perform an initial pass over the text to obtain source symbols and their frequencies. Instead, they start with an initial empty vocabulary, and read one symbol at a time. Whenever a symbol is read, it is encoded by using its current frequency distribution and its number of occurrences is increased. If a new symbol is encountered, it is added to the vocabulary. Therefore, a same symbol can be assigned different codewords during the process. The codeword of each symbol is *adapted* to its frequency as the compression progresses, but the decompressor also does the same. That is, the decompressor adapts the correspondence between symbols and codewords in the same way as the compressor does. Hence, *one-pass* techniques do not need to include the mapping between symbols and codewords along with the compressed text. This property gives to *one-pass* methods the ability to compress text streams, unlike semi-static techniques. In fact, dynamic models usually refer the encoder and decoder as *sender* and *receiver*, respectively. Ziv-Lempel family [ZL77, ZL78, Wel84], as well as PPM [CW84] and arithmetic encoding [Abr63, WNC87, MNW98] are common examples of adaptive methods.

On the other hand, a second classification can be done according to the coding process. Here we distinguish two main families:

- *Statistical techniques* : these methods assign a codeword to each source symbol whose length depends on its probability. Compression is achieved by assigning shorter codewords to more frequent symbols. Well-known statistical compressors are based on Hufmman codes, Dense Codes, and arithmetic codes.

- *Dictionary techniques* : these techniques use a *dictionary* of substrings that is built during compression. Sequences of source symbols are then replaced (encoded) by small fixed length pointers to an entry in the dictionary. Therefore, compression is obtained as long as large sequences are substituted by pointers with less space requirements. The Ziv-Lempel family holds the best known dictionary-based compression methods. Indeed, also grammar-based compressors are commonly included under this category. They are considered as a specialized form of dictionary techniques, since they operate in a similar manner to dictionary-based approaches, but generating a *context-free grammar* from which then derive the contents of the original source text. Unlike traditional dictionary-based methods, grammar-based techniques are able to faster recognize complex patterns. This makes grammar-based algorithms perform better on highly structured inputs. Re-Pair technique [LM00] is one of the best known grammar-based compressors.

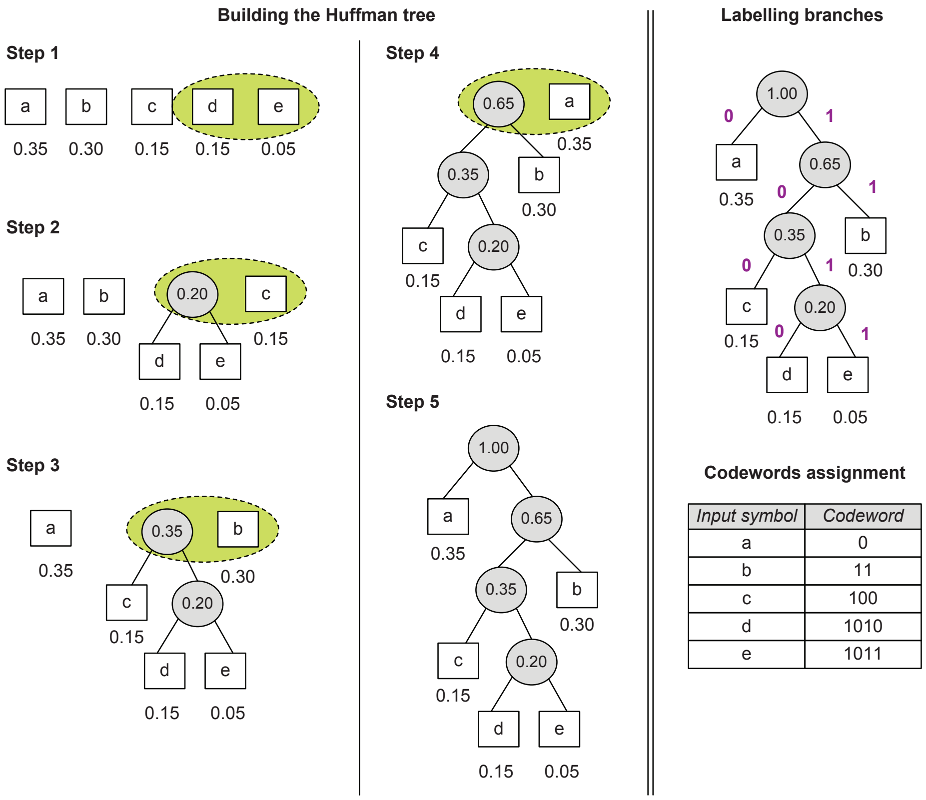
Following this last classification, sections from Section 3.1.4 through Section 3.1.6 present a brief description of some of the most interesting natural language text compression methods used nowadays.

3.1.4 Statistical Compressors

3.1.4.1 Classic Huffman Code

The classic Huffman technique [Huf52] is one of the most famous statistical semi-static text compressors. In fact, it was the first method able to generate optimal (i.e. with minimum average length) prefix free codes. The codeword generation is based on the use of a full tree, built on the first pass over the text from the different source symbols and their frequencies. As a full tree, every node of the Huffman tree has zero or D children. In case of the classical Huffman tree, $D = 2$, thus yielding to a binary tree. Each leaf node of the tree corresponds to a source symbol, and is assigned a weight given by the probability of its symbol. The position (level) of the source symbols in the tree depends on their probability. The deeper the level of the tree it is placed, the lower its probability. The Huffman tree is built in the following way. A set of nodes is first created, each one associated to a different source symbol, hence storing its corresponding frequency. Then, in a second step, the two least frequent nodes are removed from the set, leading to a new internal node, set as their parent. This new node is inserted into the set with an associated frequency computed as the sum of the frequencies of those removed nodes. Next, the same procedure is applied to the two least frequent nodes, and the whole process is repeated until just one node remains in the set. This last node constitutes the root of the Huffman tree, whose frequency is the sum of the frequencies of all the source symbols. Once the complete Huffman tree is created, codewords are assigned to source symbols (leaf nodes). By setting to 0 and to 1 the left and right branches of the internal nodes, respectively, each source symbol is mapped to a binary codeword given by the complete path from the root of the tree until that leaf node.

In Figure 3.1 an example of Huffman tree construction is depicted for the source alphabet $\beta = \{a, b, c, d, e\}$. Notice that in the first step we could choose either c or d as the second least frequent node (since both have the same frequency), together with e (which is the node with the lowest frequency). In this example we decide to choose d, and join d and e to create a new internal node, its frequency being 0.20.



Input symbol	Codeword
a	0
b	11
c	100
d	1010
e	1011

Figure 3.1: Building a classic Huffman tree.

The same occurs in step three, regarding node a and the node formed after joining the two least frequent nodes in the second step, which is finally chosen along with b, to create a new node. Finally, it is in step four that only two nodes remain, and the complete Huffman tree is created by joining them. However, observe that if we have taken a different decision regarding the joined nodes in the first and third step, a distinct Huffman tree could be obtained, thus leading also to a different encoding. That is, usually several Huffman trees can be built over the same sequence of source symbols and probabilities, generating different codes. This makes necessary for the decompressor to know the shape of the Huffman tree used during compression, which is included in a header together with the compressed text. The decompresor reads a bit at a time and traverses the Huffman tree (choosing either the right or the left branch of an internal node depending on the bit value) until a leaf is reached. At this moment, the associated source symbol is output. Then, the decompression algorithm goes back again to the root of the tree and continues the process.

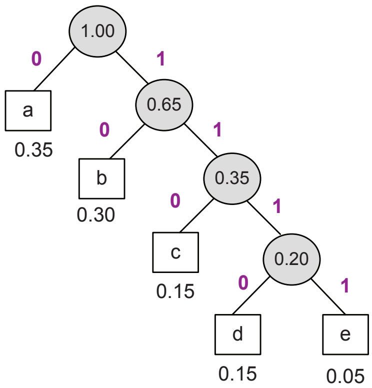


Figure 3.2: Example of canonical Huffman tree.

Canonical Huffman Tree

The main drawback of the classical Huffman code, given by the necessity of storing the tree shape used for encoding, was beaten as the concept of *canonical Huffman code* was introduced [SK64]. There, authors realized that Huffman algorithm is only needed to compute the length of the codewords. Once they are known, codewords assignment can be performed in several ways. Hence, the relevant information is just provided by the codewords length. The *canonical Huffman code* exploits that feature. It builds a prefix code tree from left to right in increasing order of depth. At each level, leaves are placed in the first position available (from left to right). Then, the following properties arise:

- Codewords are assigned to symbols in increasing length order, the lengths being given by Huffman algorithm.
- Codewords of a given length are consecutive binary numbers.
- The first codeword c_l of length l is related to the last assigned codeword, of length $l - i$, by the expression $c_l = 2^i(c_{l-i} + 1)$.

Therefore, the canonical Huffman tree can be compactly represented by only using the lengths of the codewords, which reduces the space requirements of the header of the compressed file. Figure 3.2 shows the canonical Huffman tree of the example of Figure 3.1. The codewords obtained for each source symbol are now $a \leftrightarrow 0, b \leftrightarrow 10, c \leftrightarrow 110, d \leftrightarrow 1110, e \leftrightarrow 1111$.

3.1.4.2 Plain Huffman and Tagged Huffman Codes

Huffman approaches are mostly used as character-based and bit-oriented codes. However, as stated in Section 3.1.3 using words as source symbols instead of characters greatly reduces compression ratios [Mof89]. Moreover, the use of bytes as target symbols was also explored [dMNZBY00], as a way to speed up the processing of the compressed text.

Example: to love and to be loved

Plain Huffman

word	*codeword*
to	00
love	01
and	10
be	11 00
loved	11 01

Tagged Huffman

word	*codeword*
to	10
love	11 00
and	11 01 00
be	11 01 01 00
loved	11 01 01 01

Searching for "to"

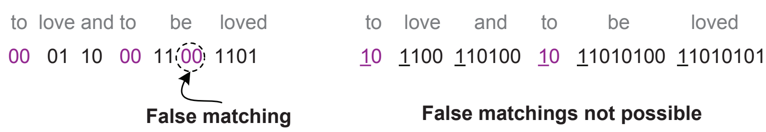


Figure 3.3: Example of false matchings in Plain Hufmman, but not in Tagged Huffman. Notice that special "bytes" of two bits are used for simplicity.

The basic word-based byte-oriented variants of the Huffman code are called *Plain Huffman* and *Tagged Huffman* [dMNZBY00]. Although compression ratios are slightly degraded with respect to a bit-oriented approach [TM97] (from 25% to 30%, in natural language text), the use of bytes provides faster decompression and searching, because no bit manipulations are necessary. The main difference between both methods is that Plain Huffman Codes do not allow searching for a pattern over the compressed text by coding it and then applying a classical string matching algorithm [BM77, NR02]. *Spourious matches* may occur (see Figure 3.3), thus a sequential search over the compressed text has to be performed, reading one byte at a time. Instead, Tagged Huffman Codes avoid that problem by introducing a simple modification in the encoding scheme: the first bit of each byte is reserved to flag the first byte of a codeword. Then, the remaining 7 bits are used for the Huffman code (since the flag is not useful by itself to make the code a prefix code). That is, full bytes are used, but only 7 bits are devoted to coding. Due to this, Tagged Huffman achieves worst compression ratios than Plain Huffman. In exchange, searches are performed much faster in the former, and it also permits random decompression.

3.1.4.3 End-Tagged Dense Code and (s,c)-Dense Code

The *End-Tagged Dense Code* (ETDC) [BINP03, BFNP07] is also a semi-static statistical word-based byte-oriented prefix-free encoder, that achieves the same search performance and capabilities of Tagged Huffman (i.e. use of string matching algorithms over the compressed text and direct access), while keeping similar compression ratios to those obtained by Plain Huffman. Hence, it combines the best properties of each of the previous alternatives. Besides, encoding and decoding with this compression technique are simpler and faster than with Tagged Huffman and Plain Huffman.

The basic idea of ETDC consists of marking the *end* of a codeword instead of the *beginning*, as Tagged Huffman does. That is, the first bit of each byte is reserved to flag whether the byte is the last one of its codeword: the highest bit of codeword bytes is 1 for the *last* byte (not the first) and 0 for the others. This simple change is enough to ensure that the code is a prefix code regardless the content of the 7 remaining bits. Therefore, unlike Tagged Huffman, ETDC does not need to use Huffman coding over the other 7 bits of each byte. Rather, *all* possible combinations of 7 bits are feasible, thus producing a *dense* encoding. This feature is the key to improve the compression ratio achieved by Tagged Huffman.

In general, we can say that for target symbols of b bits ($b = 8$ in the byte-oriented version), and given source symbols sorted by decreasing frequencies, the corresponding codewords using ETDC are formed by a sequence of symbols representing digits from 0 to $2^{b-1}-1$, except the last one which has a value between 2^{b-1} and $2^b - 1$. This codewords assignment is performed in a sequential way, thus making the computation of codes extremely simple and faster than using Huffman. Furthermore, note that a source symbol will be assigned a codeword depending on its rank in the sorted vocabulary, not on its actual frequency. As a result, no additional information is needed apart from the sorted vocabulary for the decompressor to rebuild the model[6].

If we focus on byte-wise codewords ($b = 8$), we can observe that ETDC uses 128 different values (from 0 to 127) for the symbols that do not end a codeword, called *continuers* (c), and the same amount of values, but ranging from 128 to 255, for the last symbol of the codewords, known as *stoppers* (s). However, this proportion between the number of continuers and stoppers ($s = c = 2^{b-1}$) could not be optimal for a given word frequency distribution of a text. Hence, *(s,c)-Dense Code* (*(s,c)*-DC) [BFNP03, BFNP07] is a generalization of ETDC where any $s + c = 2^b$ can be used, such that digits between 0 and $s - 1$ are used as stoppers and digits between

[6] Remember that Huffman codes need to store some information about the shape of the Huffman tree, even for the canonical tree.

Word rank	Codeword assigned	# bytes	# words
0 1 2 … s-1	[0] [1] [2] … [s-1]	1 1 1 … 1	s
s s + 1 s + 2 ... s + s - 1 s + s s + s + 1 … s + sc - 1	[s][0] [s][1] [s][2] … [s][s-1] [s+1][0] [s+1][1] … [s+c-1][s-1]	2 2 2 … 2 2 2 … 2	sc
s + sc s + sc + 1 … s + sc + sc - 1 s + sc + sc … s + sc + sc^2 - 1	[s][s][0] [s][s][1] … [s][s+c-1][s-1] [s+1][s][0] … [s+c-1][s+c-1][s-1]	3 3 … 3 3 … 3	sc^2
...	...	...	...

Figure 3.4: Codewords assignment in (s, c)-Dense Code.

s and $s + c - 1 = 2^b - 1$ are used as continuers[7]. Optimal values for s and c are computed for a specific word frequency distribution to minimize compression ratios [BFNP07]. In this way, and considering a byte-oriented scenario, the corresponding *(s,c)*-DC encoding process of a sorted vocabulary, summarized in Figure 3.4, can be described as follows:

- One-byte codewords, from 0 to $s - 1$, are given to the first s words in the vocabulary.
- Words ranked from s to $s+sc-1$ are sequentially assigned two-byte codewords. The first byte of each codeword has a value in the range $[s, s + c - 1]$, that is, a *continuer*. The second byte, the *stopper*, has a value in range $[0, s - 1]$.
- Words from $s + sc$ to $s + sc + sc^2 - 1$ are assigned three byte codewords, and so on.

For example, the codes assigned to symbols $i \in 0 \ldots 18$ by a (2,6)-DC[8] are as

[7] Notice that ETDC is actually a (2^{b-1}, 2^{b-1})-Dense Code.

[8] Note that, for simplicity, we assume bytes of 3 bits. Thus $2^3 = 8 = 2 + 6$.

follows: ⟨0⟩, ⟨1⟩, ⟨2,0⟩, ⟨2,1⟩, ⟨3,0⟩, ⟨3,1⟩, ⟨4,0⟩, ⟨4,1⟩, ⟨5,0⟩, ⟨5,1⟩, ⟨6,0⟩, ⟨6,1⟩, ⟨7,0⟩, ⟨7,1⟩, ⟨2,2,0⟩, ⟨2,2,1⟩, ⟨2,3,0⟩, ⟨2,3,1⟩, and ⟨2,4,0⟩.

In addition, there are on-the-fly procedures to encode and decode a word given its ranked position. Let i be the position of the word and $x = i - \frac{sc^{k-1}-s}{c-1}$, the first $k-1$ digits of the codeword are filled with the representation of number $\lfloor x/s \rfloor$ in base c, adding then s to each digit, and the last digit is $x \ mod \ s$.

3.1.4.4 Arithmetic Coding

Arithmetic coding [Abr63] is another example of statistical compression method, yet unlike the previous compressors, it is commonly used in an adaptive way. The main idea of this technique is to code a sequence of source symbols by using an *unique* real number in the range $[0, 1)$. That is, the algorithm starts with the initial interval $[0, 1)$, then a source symbol is read at a time, and its probability is used to narrow the interval. The obtained reduced range at each step represents the input sequence of source symbols already processed. Specifying a narrow interval requires more bits, so the number constructed by the algorithm grows continuously. To achieve compression, the interval is reduced less when a high-probability symbol is read, than when a low-probability one is processed, in such a way that most frequent symbols contribute fewer bits to the output.

To show how this compressor works, we next explain an example of arithmetic compression using a semi-static model[9]. Figure 3.5 illustrates the complete process. Let us consider a vocabulary of four source symbols $\beta = \{a, b, c, d\}$, with associated probabilities $p = \{0.4, 0.3, 0.15, 0.15\}$, and the following input message, *aabdb*. The algorithm initially divides the interval $[0, 1)$ in four subintervals according to the source symbol probabilities. Hence, the subinterval $[0, 0.4)$ represents symbol a, while any number in the subintervals $[0.4, 0.7)$, $[0.7, 0.85)$ and $[0.85, 1)$, represents symbols b, c, and d, respectively. The algorithm starts by reading the first input symbol a, thus reducing the current interval to $[0, 0.4)$. Then, this new interval is also partitioned into subintervals of different size according to the probability of the source symbols. In this case, the next possible subintervals are $[0, 0.16)$, $[0.16, 0.28)$, $[0.28, 0.34)$ and $[0.34, 0.4)$, each one representing the sequences *aa*, *ab*, *ac*, and *ad*. Since the second symbol is a again, the current interval is narrowed to $[0, 0.16)$. Next, b is read, and the interval $[0, 0.16)$ is reduced from its 40% point to its 70% point (in accordance with the probability of b). The resulting interval is $[0.064, 0.112)$. This is later narrowed by d symbol, leading to the working-interval $[0.1048, 0.112)$. Finally, the last symbol, b, reduces once more the interval to the range $[0.11092, 0.112)$. Any number of this final range could be used to represent *aabdb* message. Therefore, the encoder generates the number that could be encoded with less bits inside that interval.

[9]The dynamic version could be performed by just adapting the frequency of the source symbols each time one of them is processed.

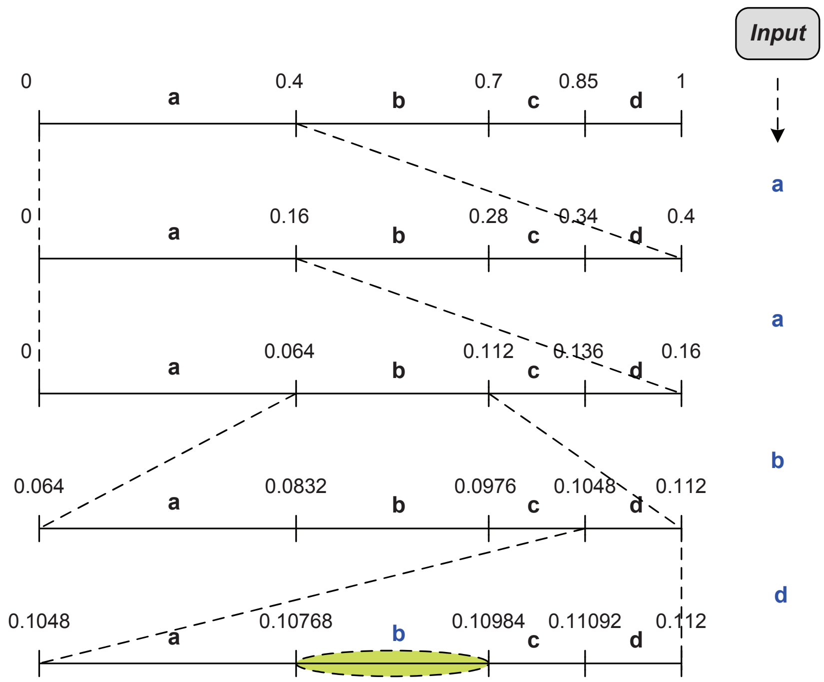


Figure 3.5: Example of arithmetic compression for the text *aabdb*.

To decompress a message coded with arithmetic coding, it is only necessary for the decompressor to know the vocabulary used and the probabilities of source symbols. From the compressed data, it is able to derive the ranges used, and hence to recover the input symbols.

Several modifications have been proposed to the basic arithmetic coding along the years, such as an integer-based arithmetic encoder [WNC87], or the use of shift/add operations instead of multiplications and divisions to improve its performance [MNW98].

3.1.4.5 Prediction by Partial Matching

Prediction by Partial Matching (PPM) is a statistical adaptive compressor [CW84], based on the use of $m + 1$ finite-context models of order from 0 to m (m is the maximum context length) to predict the probability of the next source symbol. For each finite-context model of order k (see Figure 3.6), PPM stores the different k-length sequences of characters previously encountered, and for every distinct character following those sequences, it also keeps the number of times they have appeared. These values are then used to estimate the probability of the incoming characters in that model.

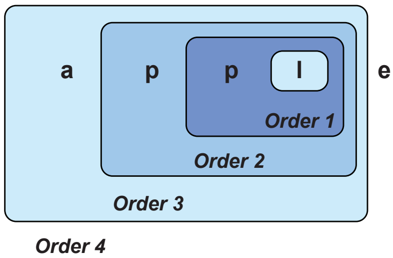


Figure 3.6: Different k-order models.

Given a source symbol s, PPM first tries to encode it by using the sequence of the m previous symbols in the input stream. That is, it starts by trying to use the probability predicted by the highest-order model (the m-order model). However, if there is no m-length sequences of characters preceding the input symbol, it means that the new character could not be encoded by the given m-order model, and the $(m-1)$-order model is then tried. In such a case, an *escape symbol* is sent to warn the decoder that a change of model will be performed. The process continues in a same way until reaching either a model for which the input character is not novel or the *bottom-level* model, which corresponds to a (-1)-order model where all symbols of the source alphabet are equally probable. In both situations, an arithmetic coding is applied to encode the incoming symbol using the predicted probability of the attained model.

Different methods to determine how probabilities are assigned to the *escape symbols* rise also distinct variants of PPM. They are usually denoted as PPMu, where u, indicates the method used. For instance, methods A, B [CW84], C [Mof90], D [How93], and X [WB91], have been proposed, and further compared [MT02].

In general, and given their nature, PPM algorithms achieve good compression ratios, at the expense of worsening compression and decompression speed. For instance, PPM obtains compression ratios around 20%-25% when working with natural language texts, while in character-based Huffman codes this value grows up to 65%. However, unlike Huffman codes, where the use of words could significantly improve compression performance, word-based PPM models are unfeasible in practice. Since word-based vocabularies are larger than character-based ones, to keep context models of order greater than 2 becomes impracticable.

3.1.5 Dictionary-based Compressors

3.1.5.1 Ziv-Lempel Family

Among the dictionary-based compressors, Ziv-Lempel family includes the most representative dictionary and adaptive techniques. Some well-known variants of this family are LZ77 [ZL77] and LZ78 [ZL78] algorithms, which are the basis of commonly used compressors such as *gzip*[10], *compress* and *p7zip*[11].

Unlike other adaptive approaches, such as Arithmetic Coding or even PPM, which have proven to be competitive techniques regarding compression ratio, Ziv-Lempel compressors do not achieve as good compression values (around 35%-40%). In exchange, their main advantage is compression and, specially, decompression speed.

LZ77. LZ77 is the first proposed compression method of the Ziv-Lempel family. The main component of this technique is a fixed length *sliding window* holding the n last characters already processed. Therefore, the basic idea of LZ77 is to perform a dictionary strategy, based on the use of the *sliding window*, to code next input symbols. The process starts with an *empty* window. In each step, the algorithm reads the largest substring, t, already appeared in the window. Let us assume that $t = t_0, t_1, \ldots, t_{l-1}$, and that c is the next incoming character after t. Then, LZ77 encodes that sequence as a triplet $\langle p, l, c \rangle$, where the p value denotes the backward offset with respect to the end of the window where t starts, and l represents the length of the t substring. Next, the generated triplet is output and the window is slid by $l + 1$ positions. In case no substring is found in the window, the transmitted triplet is $\langle 0, 0, c \rangle$ and the window is slid only one position. Figure 3.7 shows an example of compression using LZ77 technique, for the text *abbabcabbbbc*. Shaded characters represent the current *sliding window* at each step.

Notice that by using this scheme, decompression can be performed very fast. During decompression, the window holds the last decoded characters. Hence, given a triplet $\langle p, l, c \rangle$, the decoder only needs to output l characters, starting at position p before the end of the window, next followed by c.

LZ77 performance mainly depends on the size of the *sliding window*. The greater the size of the window, the greater the probability to encode larger substrings. However, the range of values needed to represent p offset also growths as the size of the window increases. Usually, 12 bits are used to represent p (thus yielding a *sliding window* of 4096 bytes), and 4 bits, for l. That is, both p and l are represented by using 2 bytes. Furthermore, a minimum size of the window is also considered in order to avoid the replacement of small prefixes, that would not pay off the triplet.

[10] http://www.gzip.org
[11] http://www.7-zip.org

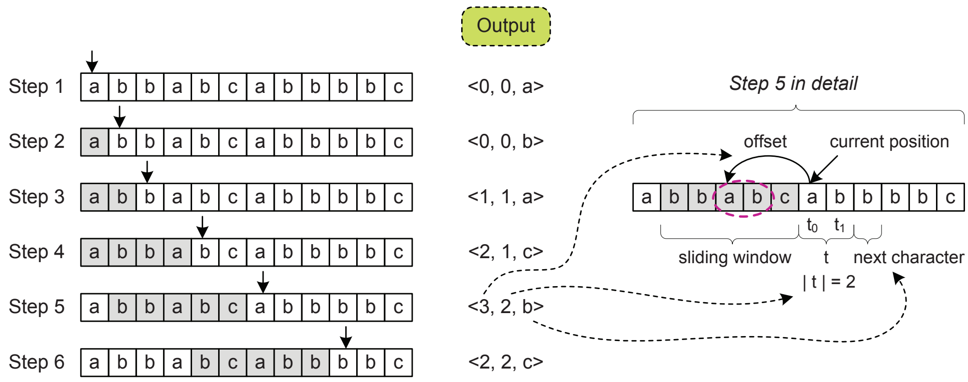
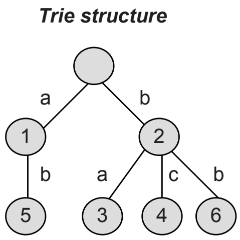


Figure 3.7: Compression of the text *abbabcabbbbc* using LZ77.

The LZ77 technique constitutes the base of *gzip* compressor[12]. A well-known variant of LZ77 is LZMA (*Lempel-Ziv-Markov chain algorithm*). This variant commonly uses a dictionary size of 1GB, although this value can growth up to 4GB, if required. *p7zip* is a compression tool based on LZMA algorithm. In general, it achieves better compression ratios than *gzip* (around 22%-30%), but compression/decompression process suffers from large memory and time requirements.

LZ78. The second proposal of the Ziv-Lempel family is the LZ78 compressor. This technique substitutes the use of the *sliding window* by a dictionary that holds all the appeared substrings in the source text. This dictionary is efficiently searched via a *trie* data structure. Each node n_i of the *trie* is a pointer to a dictionary entry, $entry_i$, representing the substring obtained by the concatenation of letters labeling the *trie* edges in the path from the root of the *trie* to node n_i. In this way, a character is read at a time, traversing the *trie* downwards until the longest matching entry ($entry_k$) is found (that is, until there is no edge that permits the transition to the next incoming character of the text). The read sequence is then encoded by a pair $\langle k, c\rangle$, where k is the pointer to the found dictionary entry, $entry_k$, and c is the character of that follows $entry_k$ in the text. This pair is sent to the output and the substring $entry_k + c$ is appended to the dictionary as a new entry. The LZ78 decompressor works in a similar way than the encoder, but traversing the *trie* upwards. We illustrate an example of text compression with LZ78 technique in Figure 3.8. We use the same input as that used to exemplify LZ77 compressor: *abbabcabbbbc*.

[12]It is based on the compression algorithm known as *deflate* which uses a combination of LZ77 and Huffman coding.



Step	Input	Output	Dictionary
1	a	(0, a)	$entry_1$ = “a”
2	b	(0, b)	$entry_2$ = “b”
3	ba	(2, a)	$entry_3$ = “ba”
4	bc	(2, c)	$entry_4$ = “bc”
5	ab	(1, b)	$entry_5$ = “ab”
6	bb	(2, b)	$entry_6$ = “bb”
7	bc	(4, ε)	

Figure 3.8: Compression of the text *abbabcabbbbc* using LZ78.

Usually, LZ78 compression is faster than LZ77. However, LZ78 decompression speed is not as good as that obtained by LZ77. Yet, a variant of LZ78, called LZW [Wel84], is widely used. For instance, it is the base of the Unix *compress* program, and also of *GIF* image format. The main difference of LZW with respect to LZ78 is that LZW only outputs pointers to found dictionary entries, but not characters, as LZ78 does. To this aim, LZW initializes the dictionary with the symbols that compose the source alphabet, and takes the last character of the just previously found substring as the first character of the next one.

3.1.5.2 Re-Pair

Re-Pair [LM00] is an example of grammar-based compressor. It consists of repeatedly finding the most frequent pair of symbols in a sequence of integers, and replacing it with a new one, until no more substitutions are useful. Basically, given a sequence T, this technique first identifies the most frequent pair ab in T. Then, a rule $r \to ab$ is generated and appended to a dictionary D, r being a new symbol not appearing in T. In such a way, every pair ab in T is next replaced by r. This whole process is repeated until there is no pair of adjacent symbols that occurs twice. Therefore, if we denote as C the sequence obtained after T is compressed, any symbol in C is either an original symbol of T (called a *terminal*) or an introduced symbol (called a *non-terminal*). Each of them represents a substring of T of length 1 or longer than 1, respectively. In case of a *non-terminal* symbol, the original substring can be recovered by recursively expanding that symbol. That is, any symbol r can be expanded using rule $r \to r^1 r^2$ in D, and the process continues in a same way with r^1 and r^2, until the original symbols of T are obtained. Any substring is expanded in optimal time (i.e. proportional to its length).

Despite its quadratic appearance, Re-Pair can be implemented in linear time [LM00], but at the expense of using several data structures needed to trace the pairs that must be replaced. That may become problematic in case of large sequences

[Wan03, GN07, CN07], since the space consumption of the linear time algorithm is about $5|T|$ words.

3.1.6 Other Compressors

3.1.6.1 Burrows-Wheeler Transform (BWT)

The Burrows-Wheeler Transform (BWT) [BW94] is not an actual compressor, but an algorithm that permits to transform a string, W, into another string, TW, that is more compressible. Basically, TW contains the same data of W, but in different order. This algorithm is also reversible. That is, the original source can be recovered from TW (and little extra information).

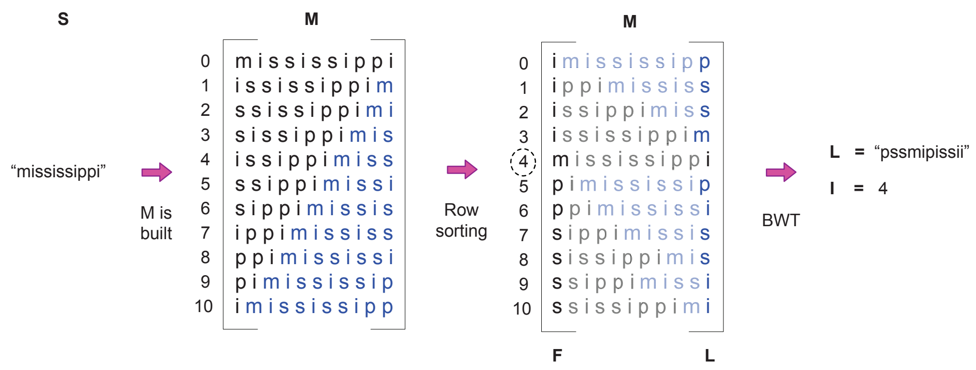


Figure 3.9: Direct Burrows-Wheeler Transform.

Direct BWT. Given a string W of length n, BWT first builds a matrix M_{nxn}, obtained from the circular rotation of the input string. That is, the first row contains W, the second one, $W >> 1$ (i.e. W circularly shifted one position to the right), and so on. Next, M_{nxn} is lexicographically ordered by row. Note that one of the rows of the sorted matrix stores the original string W. Let us refer to that row as I. We also denote as F and L the first and last column, respectively, of the matrix obtained after sorting. The following properties hold:

- F contains all characters in W, but alphabetically ordered.
- The j^{th} character in L precedes (in W) the string stored at row j.

The result of applying BWT to W, is a string formed by all characters in the column L of M, plus the value I: $BWT(W) \rightarrow (L, I)$. Figure 3.9 shows how BWT works given the input string *mississippi*.

Inverse BWT. The inverse BWT uses the result obtained by BWT algorithm (i.e. (L, I)), and recovers the original string. The first step consists of rebuilding the column F of matrix M, by simply sorting alphabetically the string L. Both strings are used in a second step to construct a new string T that stores the correspondence between the characters of the two previous strings. That is, if $L[j]$ stores the k^{th} occurrence of the character 'c' in L, then $T[j] = i$, given that $F[i]$ is the k^{th} occurrence of 'c' in F. According to the definition of T, it also rises $F[T[j]] = L[j]$. Hence, the last step recovers the input string W using the I value, and the vectors L and T. The recovering procedure starts by doing:

$$\begin{array}{rcl} p & \leftarrow & I \\ i & \leftarrow & 0 \end{array}$$

Then, each of the n characters of W are recovered by applying n times the following procedure:

$$\begin{array}{rcl} S_{n-i-1} & \leftarrow & L[p] \\ p & \leftarrow & T[p] \\ i & \leftarrow & i+1 \end{array}$$

BWT in Text Compression. To understand why BWT obtains a more compressible representation, let us consider a text W where the word '*better*' appears many times. After the rows of M are sorted, all those rows starting with '*etter*' are placed together. Moreover, most of them are likely to end (i.e. be preceded by) in '*b*'. That is, a region of L will hold a large number of occurrences of character '*b*', in addition to some other characters that could precede '*etter*'. For instance, the characters '*l*' or '*f*', for words '*letter*' and '*fetter*', respectively. The same property can be extended to any other substring of W, in such a way that specific regions of L will contain a large number of a few distinct characters.

This result leads to the fact that a given symbol will appear with high probability in some regions of L, while in some other its probability will fall down. This feature can be efficiently profited by *move-to-front* (MTF) compressor [BSTW86], that will encode the occurrences of '*b*' as the number of distinct characters found since the last previous occurrence of '*b*'. Hence, all contiguous occurrences of any character will become sequences of consecutive zeros. What is more, the obtained representation after applying the MTF encoder, can be still further compressed using either a Huffman-based or an arithmetic encoder. However, since we could find many long runs of zeros in the output of MTF, another good alternative is to use *Run Length Encoding* (RLE). The *bzip2* compressor is based on BWT compression.

3.1.7 Measuring the Efficiency of Compression Techniques

To measure the efficiency of different compression methods, we must consider two main features. On the one hand, the performance of the algorithms involved, and on the other hand, the compression achieved. Although compression and decompression algorithms can be analyzed by their theoretical complexity, thus providing an idea of how a technique will behave, it is also relevant to compare their performance against other methods in real scenarios, based on empirical results. *Compression* and *decompression times* (measured in seconds or milliseconds) are the most usual measures used to give us this kind of information. In turn, and regarding the compression obtained by the technique, a measure commonly used is the *compression ratio*. Let us consider that z is the size of the source text in bytes, and that the compressed text occupies m bytes, then the compression ratio is defined as $\frac{m}{z}$ x 100. That is, it represents the percentage that the compressed text occupies with respect to the original text size.

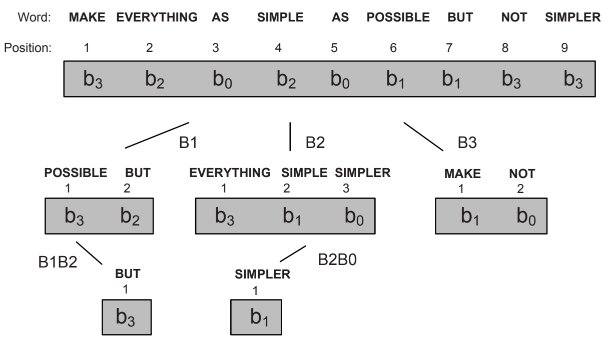
3.1.8 One Step beyond Text Compression

Word-based byte-oriented compression techniques have been acknowledged as quite relevant solutions for natural language text databases, since they achieve competitive compression ratios, fast random access, and direct sequential searching. In case of semi-static statistical methods, compression has gone one step beyond. Recently, a novel reorganization proposal of the codeword bytes of any natural language text compressed with an encoding scheme of this category has been presented [BFLN08, BFLN12]. This codeword rearrangement, called Wavelet Trees on Bytecodes (WTBC), for its similarity with the original wavelet trees [GGV03], consists basically of placing the different bytes of each codeword at different nodes of a tree, instead of sequentially concatenating them, as in a typical compressed text. However, this minor change leads to a new implicitly indexed representation of the compressed text, where search times are drastically improved, by using a negligible amount of additional space. In fact, in [BFLN12], experimental data shown that WTBC not only performs much more efficiently than sequential searches over compressed text, but also than explicit inverted indexes when little extra space is used. WTBC specially succeeds when searching for single words and short phrases. This structure has provided the inspiring starting point of this thesis work. We next conceptually describe it in detail.

The essence of this codewords rearrangement is the following: the root of the WTBC is represented by all the first bytes of the codewords, following the same order as the words they encode in the original text. That is, let us assume we have the text words $\langle w_1, w_2 \dots w_n \rangle$, whose codewords are $cw_1, cw_2 \dots cw_n$, respectively, and let us denote the bytes of a codeword cw_i as $\langle cw_i^1 ... cw_i^m \rangle$ where m is the size of the codeword cw_i in bytes. Then the root is formed by the sequence of bytes

$\langle cw_1^1, cw_2^1, cw_3^1 ... cw_n^1 \rangle$. At position i, we place the first byte of the codeword that encodes the i^{th} word in the source text, so notice that the root node has as many bytes as words has the text.

We consider the root of the tree as the first level. Therefore, second bytes of the codewords longer than one byte are placed in nodes of a second level. The root has as many children as different bytes can be the first byte of a codeword of two or more bytes. For instance, in a $(190, 66)$-DC encoding scheme, the root will have always 66 children, because there are 66 bytes that are *continuers*. Each node X in this second level contains all the second bytes of the codewords whose first byte is x, following again the same order of the source. That is, the second byte corresponding to the j^{th} occurrence of byte x in the root, is placed at position j in node X. Formally, let us suppose there are f words coded by codewords $cw_{i_1} ... cw_{i_f}$ (longer than one byte) whose first byte is x. Then, the second bytes of those codewords, $\langle cw_{i_1}^2, cw_{i_2}^2, cw_{i_3}^2 ... cw_{i_f}^2 \rangle$, form the node x in the second level. The same idea is used to create the lower levels of the tree. Looking into the example, and supposing that there are d words whose first byte codewords is x and whose second one is y, then node XY is a node of the third level, child of node X, and it stores the byte sequence $\langle cw_{j_1}^3, cw_{j_2}^3, cw_{j_3}^3 ... cw_{j_d}^3 \rangle$ given by all the third bytes of that codewords. Those bytes are again in the original text order. Therefore, the resulting tree has as many levels as bytes have the longest codewords.



SYMBOL	FREQUENCY	CODE
AS	2	b_0
POSSIBLE	1	b_1b_3
SIMPLE	1	b_2b_1
EVERYTHING	1	b_2b_3
NOT	1	b_3b_0
MAKE	1	b_3b_1
BUT	1	$b_1b_2b_3$
SIMPLER	1	$b_2b_0b_1$

Figure 3.10: Example of WTBC structure.

To better understand this reorganization of codewords Figure 3.10 shows an example where a WTBC is built from the text `MAKE EVERYTHING AS SIMPLE AS POSSIBLE BUT NOT SIMPLER`, and the alphabet $\Sigma = \{$`AS`, `BUT`, `EVERYTHING`, `MAKE`, `NOT`, `POSSIBLE`, `SIMPLE`, `SIMPLER`$\}$. Once codewords are assigned to all the different words in the text, by using any word-based, byte-oriented semi-static statistical compressor, their bytes are spread in a tree following the reorganization of bytes explained. That is, all the first bytes of the words are placed in the root following the text order, while the remaining bytes are in the corresponding nodes of consecutive levels. For example, b_3 is the 9^{th} byte of the root because it is the first byte of the codeword assigned to `'SIMPLER'`, which is the 9^{th} word in the text. In turn, its second byte, b_1, is placed in the third position of the child node $B3$ because `'SIMPLER'` is the third word in the root having b_3 as first byte. Likewise, its third byte, b_2, is placed at the third level in the child node $B3B1$, since the first and second byte of the codeword are b_3 and b_1, respectively. Observe that only the shaded byte sequences are stored, the rest of the text is only shown for comprehensibility.

Notice that the amount of space needed for all the nodes of a WTBC representation, matches the size of the text compressed with the compression method used to create the WTBC structure. That is, just a reorganization of the codewords bytes is performed in WTBC. Yet, this *simple* codewords rearrangement, provides important implicit indexing properties, which have a *definite* impact over the searching capabilities of this structure [BFLN12].

3.2 Succinct Data Structures

Succinct data structures aim to represent data (e.g. trees [Jac89, MR97, FM11], texts [GGV03, FM05], strings [GGV03, GMR06, HM10], graphs [Jac89, MR97, FM08a], etc.) by reducing space requirements as much as possible (close to the information theoretic lower bound), while still being able to efficiently solve the required operations over the data. Their growing interest lies in the increasing performance gap between successive levels in the memory hierarchy, since the reduction of space obtained by these structures allows them to operate on faster levels. This section discusses some of the most relevant succinct data structures, used to improve the efficiency of other high-level structures.

3.2.1 Rank and Select Data Structures

One of the first presented succinct data structures consisted of bit-vectors supporting *rank* and *select* operations [Jac89]. These basic operations constitute the basis of other important succinct data structures. We discuss them more in detail in Section 3.2.1.1. Section 3.2.1.2 also describes some solutions to support these rank and select operations over arbitrary sequences.

3.2.1.1 Rank and Select over Bitmaps

Let be $B[1,n]$ a binary sequence of size n. Then *rank* and *select* are defined as (see Figure 3.11):

- $rank_b(B,p) = i$ if the number of occurrences of the bit b from the beginning of B up to position p is i.

- $select_b(B, i) = p$ if the i^{th} occurrence of the bit b in the sequence B is at position p.

Given the importance of these two operations in the performance of other succinct data structures, specially in *full-text indexes* [NM07], many strategies have been developed to efficiently implement *rank* and *select* [MN07].

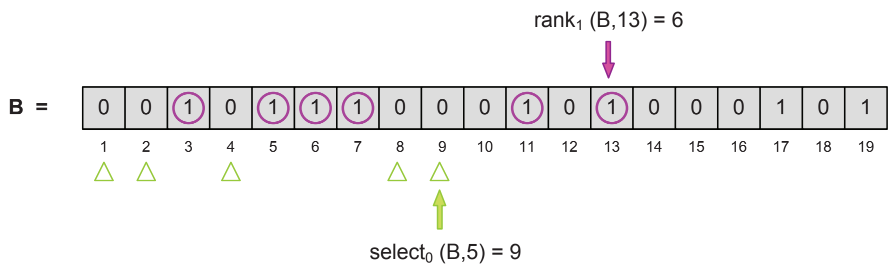


Figure 3.11: Example of *rank* and *select* operations.

As previously stated, *rank* and *select* operations were first introduced by Jacobson [Jac89]. He proposed an implementation for *rank* and *select*, able to compute *rank* in constant time. It is based on a two level directory structure. The first level directory stores $rank_b(B, p)$ for every p multiple of $s = \lfloor log\ n \rfloor \lfloor log\ n/2 \rfloor$. The second level directory holds the same information but for every p multiple of $b = \lfloor log\ n/2 \rfloor$, within each block of size s. Hence, we can compute $rank_1(B, p)$ by taking from the first level directory, the number of times the bit 1 appears until the beginning of the block of size s that contains the position p, and then adding to this value, that kept in the second level. Yet the final result is obtained by using further *table lookups*. That is, it remains to count the number of occurrences of bit 1 from the beginning of the block of size b, where position p is contained, until the position p itself. To this aim, this bit subsequence is used as the index for a table that indicates the number of occurrences of bit 1 (likewise for bit 0) in it. As a result, *rank* can be computed in constant time. Notwithstanding, binary searches are needed to calculate *select*, thus it is computed in $O(log\ log\ n)$. The overall space required by the auxiliary dictionary structures is $o(n)$.

Later works by Clark [Cla96] and Munro [Mun96] obtained constant time complexity also for *select* operation, using additional $o(n)$ space. For instance, Clark proposed a new three-level directory structure, where the first level records the positions of every $\lceil log\ n \rceil \lceil log\ n\ log\ n \rceil$'th 1 bit, and the second and third level, store the positions of bits set to 1 in the subranges corresponding to the first and second level, respectively. An analogous structure should be built to answer $select_0$.

None of the previous implementations take into account the content of the binary sequence nor its statistical properties (e.g. number of 1 bits ande their positions in the sequence) to efficiently compute *rank* and *select*. [Pag99, RRR02, OS07] are some examples of works devoted to also solve *rank* and *select*, but using representations that store a compressed form of B. Pagh proposal [Pag99] splits the binary sequence into compressed blocks of the same size, each of which is represented by the number of 1 bits it stores, and the number corresponding to that particular subsequence. The main drawback of this basic approximation lies in the almost linearly growth of the number of compressed blocks, hence an interval compression scheme is also proposed to reduce the number of compressed units by clustering suitable adjacent blocks together into intervals of varying length. Raman *et al.* [RRR02] presented a numbering scheme to represent the compressed binary sequence, in such a way that each of the blocks of size $u = \frac{log\ n}{2}$, in which the sequence is divided, is designated by a pair (c_i, o_i), where c_i indicates the number of 1 bits it contains (the *class* of the block), and o_i, represents the *offset* of that block inside a list of all the possible blocks with c_i 1 bits. In this way, blocks with few (or many) 1s require shorter identifiers and zero-order compression is achieved. This approach is currently the best complete representation of binary sequences [MN07] (since it supports *rank* and *select* in constant time for both 0 and 1 bits), yet it is not anymore simple to implement. Further works such as those introduced by Okanohara and Sadakane [OS07], were devoted to propose several practical alternatives achieving very close results based on different rank/select directories: *esp*, *recrank*, *vcode*, *sdarray*, and *darray*. Each variant has different advantages and drawbacks (regarding its size and time-complexity) since different ideas are behind each one. Most of them are very good for *select* operations, but *rank* queries are commonly slower.

Another alternative study, called *gap encoding*, aims to compress the binary sequences when the number of 1 bits is small. It is based on encoding the distances between consecutive 1 bits. Several developments following this approach have been presented [Sad03, GGV04, BB04, GHSV06, MN07].

3.2.1.2 Rank and Select over Arbitrary Sequences

Although *rank* and *select* operations were initially defined over binary sequences, they have also been proved to be necessary operations over sequences of symbols of an arbitrary alphabet, Γ. In such a case, given a sequence of symbols $S = s_1 s_2 \ldots s_n$, and a symbol $s \in \Gamma$, *rank* and *select* can be described as:

- $rank_s(S, p) = i$ if s appears i times in the sequence up to position p.
- $select_s(S, i) = p$ if p is the position of the sequence containing the i^{th} occurrence of the symbol s.

In this general scenario, the strategies proposed for binary sequences cannot be directly applied. Therefore, the computation of *rank* and *select* over arbitrary sequences is usually tackled by reducing the problem to the use of bit-oriented *rank* and *select* operations.

Bitmaps. The simplest approach to answer *rank* and *select* operations over an arbitrary sequence of symbols consists of using a bitmap for each symbol $s \in \Gamma$, in such a way that the positions of a symbol bitmap corresponding to the positions of the original sequence where the specific symbol appears are set to 1. Since *rank* and *select* operations over binary sequences can be answered in constant time, it will be also the same for arbitrary sequences, if we follow this approach. Still, the main drawback of this solution is the space required for the bitmaps, plus that needed for the auxiliary structures to compute *rank* and *select* in constant time in each one of them.

Wavelet Trees. A *wavelet tree* [GGV03] is a structure that allows efficiently computing *rank* and *select* over arbitrary sequences of symbols. It consists of a balanced binary tree storing a bitmap in each node. The root of the tree contains a bitmap of size n (being n the length of the sequence), where the positions holding an occurrence of a symbol belonging to the first half of the alphabet Γ, are set to 0, and 1, in the other case. Then, those symbols given a 0 in that bitmap are processed in the left child node, while the rest are processed in the right child. The same procedure is applied in both children, and recursively repeated until the alphabet cannot be divided, thus reaching the leaves of the tree. In this way each node indexes half the symbols (from Γ) indexed by its parent node. In Figure 3.12 an example of how the wavelet tree is built from a sequence of symbols over the alphabet $\Gamma = a, b, c, d$ is depicted[13].

By using this structure there is no need to store the original sequence separately. It can be recovered from the bitmaps. Furthermore, it is extremely simple to compute *rank* and *select*, through top-down and bottom-up traversals of the wavelet tree, respectively. For instance, let us assume that we want to compute $rank_a(S, 6)$ in the example of Figure 3.12. As symbol a belongs to the first half of the alphabet, we know that it is associated with 0 bit occurrences in the bitmap of the root node and that it will be further processed in the left child. Hence, we first compute a binary *rank*, $rank_0(B_1, 6) = 2$, over the root bitmap and then we move to the left child. The obtained result, 2, is used again to perform a new binary *rank* in the second level. Note that, in this level, a is also represented with a 0 bit, so we calculate $rank_0(B_2, 2) = 2$. Given that we are in the last level of the tree, this value also indicates the final answer to $rank_a(S, 6)$, that is 2. To compute *select*, a similar procedure is performed, but starting from the leaf nodes and moving up

[13] Notice that each node only stores the bitmap, the rest of the text is only shown for clarity.

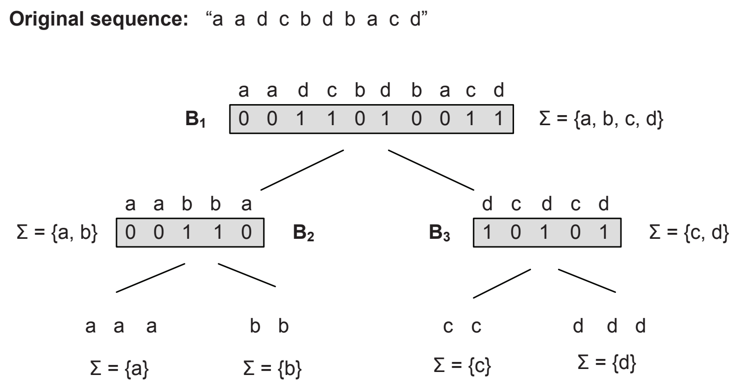


Figure 3.12: The wavelet tree of the sequence *aadcbdbacd*.

to the root of the tree. For example, let us suppose we want to know the position where the 2^{nd} occurrence of c is placed in the sequence S. Each symbol $s \in \Gamma$ is associated with an unique leaf node in the tree[14], thus the *select* procedure will start, in this case, at node B_3. Since c is represented with 0 bits in that node, there we compute $select_0(B_3, 2) = 4$. Moreover, node B_3 is the right child of its parent node, B_1, (that is, all symbols represented in node B_3 come from 1 bits in node B_1), so with the obtained result we know that the second occurrence of c is the fourth 1 bit in B_1. In this way, we next compute $select_1(B_1, 4) = 9$, and then we can finally answer that $select_c(S, 2)$ is 9.

Practical variants of the wavelet tree achieve zero-th order entropy by giving to the tree the shape of the Huffman tree of the sequence, or use Raman *et al.* data structures for rank/select operations [GGV03, NM07].

Golynski et al. Solution. These authors [GMR06] proposed a data structure able to answer *rank* operations in time $O(log\ log\ \sigma)$, σ being the size of the alphabet Γ, and *select*, in $O(1)$. The main idea is to reduce the problem over one sequence of length n, and alphabet σ, to n/σ sequences of length σ. Given a sequence S, a table T of size σxn is built to represent the sequence, where those rows are indexed by $1, \ldots, \sigma$, and columns, by positions in the sequence (i.e. from 1 to n). Each entry $T[s, i]$, takes value 1 if symbol $s \in \Gamma$ occurs in position i in the sequence, and 0 otherwise. Let A be a bitmap of length $\sigma \cdot n$ obtained by writing T in row major order. Then A is split into blocks of size σ, in such a way that *rank* and *select* are answered over these blocks by defining and implementing restricted versions of those operations. Since the space required by A is too high, it is not actually stored.

[14]This leaf node is determined by the position of the symbol in the alphabet.

Instead, a new bitmap B is created containing the cardinalities (i.e. number of 1s) of every block of A, in unary. Assuming that k_i is the cardinality of block i, then $B = 1^{k_1}01^{k_2}0 \ldots 1^{k_n}0$. Yet, with B we can answer *rank* only for positions that are multiples of σ, and we can only determine in which block is the $i-th$ occurrence, for *select* (by means of restricted *rank* and *select* operations). Therefore, we still need to examine the blocks. Each of them is represented by using two sequences. On the one hand, a bitmap called X, stores the cardinality of every symbol s in the block, using the same encoding as for B. On the other hand, a sequence π, indicates the positions of all the occurrences for each symbol s in the block, in alphabetical order. That is, π, stores the permutation obtained by stably sorting the sequence represented by the block. With these additional data structures, *rank* and *select* operations can be answered also inside the blocks.

This thesis specially focuses on the problem of computing *rank* and *select* over sequences of bytes, as it will be further discussed in Section 5.2. For this particular case, it has been shown [Lad11], the good performance of an implementation obtained by adapting the Jacobson proposal [Jac89].

Byte-oriented Rank and Select Solution. This approach [Lad11] is based on a two-level directory structure of partial counters to avoid counting the number of occurrences of a searched byte from the beginning of the sequence. Given a sequence of bytes $B = b_1, b_2 \ldots b_n$, it is divided into chunks of size sb and bl, called *superblocks* and *blocks*, respectively. For each byte b, the first level contains the number of times it appears from the beginning of the sequence up to the start of each superblock. In turn, the second level stores the number of occurrences of each byte until the start of each block, but from the beginning of the superblock it belongs to. With this additional structure, $rank_b(B,p)$ can be computed by taking the values recorded for byte b into the corresponding superblock and block where p takes place, and then adding the number of occurrences of byte b from the beginning of the specific block to position p itself. Hence, finally, *rank* can be answered in time $O(bl)$. For instance, Figure 3.13 shows an example of how to compute $rank_{13}(B, 317)$, through this scheme. Note that the position $p = 317$ is hold into the *superblock* 2 (sb_2) and, more precisely, into *block* 7 (bl_7). Therefore, we just need to add the values stored for byte $b = 13$, in the corresponding counters (that is, $sb_2[13] = 15$ and $bl_7[13] = 3$, respectively), plus the appearances of byte 13 from position 301 (i.e. the beginning of block 7) until position 337 (we can see in Figure 3.13 that byte 13 appears 2 times inside that range). In this way, we obtain the final answer, $rank_{13}(B, 317) = 15 + 3 + 2 = 20$. With respect to $select_b(B,p)$, a binary search is first performed inside the values stored in the superblocks, followed by an additional search in the blocks of the found superblock. The final step consists of a sequential scan in the obtained block. This procedure rises a time $O(bl + log\ n)$.

There is a tradeoff between space and time. The more partial counters (i.e. the

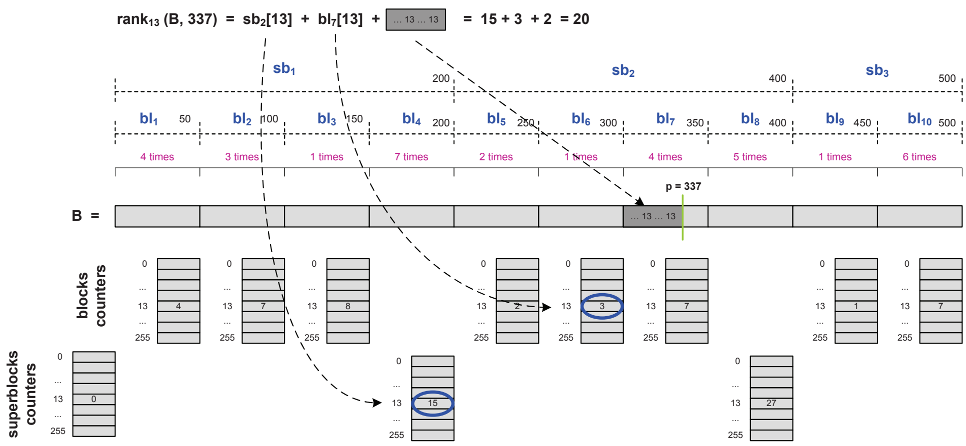


Figure 3.13: Example of byte-oriented *rank* operation by using a two level-directory structure of partial counters.

shorter bl), the more the space needed, but the more efficient the rank and select operations (i.e. the faster the sequential counting of occurrences of byte b).

3.2.2 Succinct Tree Representations

Trees are one of the most important data structures. Given a general tree of n nodes, a classical representation uses $O(n)$ pointers (or words), each one requiring $w \geq log\ n$ bits, thus leading to $O(nw)$ bits of space. The associated constant is at least 2, which permits to support basic operations such as moving to the first child and to the next sibling, or to the i^{th} child. Some other simple operations (e.g. moving to the parent, obtaining the depth, etc.) and sophisticated ones (e.g. moving to a specific level-ancestor or to the lowest common ancestor of two nodes), are also supported, but by further increasing this constant. Therefore, along the years several works have been devoted to the problem of reducing the space needed to represent trees [Jac89, MR01, MRR01, MR04, GRR04, GRRR04, CLL05, BDM$^+$05, FLMM05, GRRR06, DRR06, BHMR07, HMR07, GGG$^+$07, Sad07, JSS07, LY08, FM08b, SN10], achieving $2n + o(n)$ bits of space and constant time for most of the operations. The main differences among the distinct proposals are mainly given by their different functionality (e.g. some works only support basic operations [Jac89, DRR06], while some others are able to answer a full range of operations [BDM$^+$05, JSS07, FM08b, SN10]), and the nature of the $o(n)$ space overhead (ranging from $O(n/(log\ log\ n)^2)$ [LY08] to $O(n/polylog(n))$ [SN10]).

Tree representations can be roughly divided into three categories. Figure 3.14 shows an example of each type of representation for a given tree:

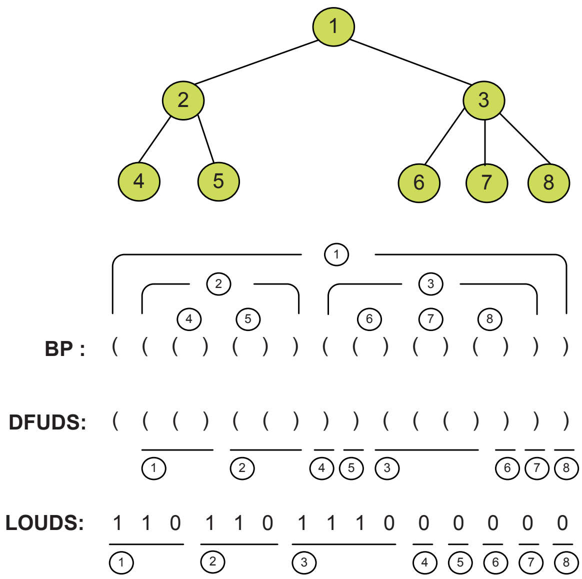


Figure 3.14: Succinct representations of trees.

- **BP**: the *balanced parentheses* representation is built from a depth-first preorder traversal of the tree, writing a '(' when arriving to a node, and a ')' when we leave it (that is, after its subtree). In this way, each node is represented by a pair of matching opening and closing parenthesis, leading to a sequence of $2n$ balanced parentheses. This representation was first advocated in [Jac89], achieving later constant times [MR01] for some core operations (e.g. *findclose*, *findopen* and *enclose*) used to solve basic tree operations (e.g. *parent*, *subtreesize*, *nextsibling*, etc.). Recently, a new proposal [SN10], has demonstrated to be able to solve in constant time many other sophisticated operations that are not usually handled by other BP representations, such as *child*, *lowest common ancestor* or even *level ancestor*.

- **DFUDS**: the *depth-first unary degree sequence* [BDM$^+$05, JSS07] is built by following the same depth-first preorder traversal as BP, but in this case, each time we arrive to a node, we write as many '(' as the number of children it has, and only one ')'. By appending an initial opening parenthesis, the resulting sequence turns out to be a balanced sequence of $2n$ parentheses. The above mentioned core operations on parentheses (i.e. *findclose*, *findopen* and *enclose*) are also used by DFUDS to support the basic functionality of classical BP representations [MR01], but in a different way. Some sophisticated operations, such as *child*, are supported as well by DFUDS, in constant time, requiring extra structures.

- **LOUDS**: the *level-ordered unary degree sequence* [Jac89, DRR06] is obtained by traversing the tree in level order and writing the degree of each node in unary. For instance, a node with 3 children will be represented as 1110. The obtained sequence has n 0's and $n-1$ 1's. Unlike the previous representations, *rank* and *select* operations over symbols '(' and ')' are just needed by LOUDS to answer a few, but key operations, such as *parent* and *child* in constant time. Yet it does not efficiently support most of the others operations.

In this thesis, we use the recent proposal called *fully-functional succinct tree* (FF) [SN10], based on a BP representation. It has been proved to be an outstanding solution that combines wide functionality, with little space usage and good time performance. We will now describe it in more detail.

3.2.2.1 Fully-functional Succinct Tree

The main component of this representation is a novel data structure, called *range min-max tree*. Just with this data structure, it is possible to answer in constant time not only the core operations, but also the complex ones. This approach differs from previous works, in which each operation needs distinct auxiliary data structures to be solved [MRR01, MR04, CLL05, Sad07, LY08].

The *fully-functional succinct tree* proposal reduces the large number of relevant tree operations considered in the literature to a few primitives that are efficiently carried out by the *range min-max tree*. Let $P = [0 \ldots n-1]$ be a balanced parentheses sequence representing a tree, and $excess(i) = rank_{(}(i) - rank_{)}(i)$, a function that gives us the difference between the numbers of opening and closing parenthesis in $P[0 \ldots i]$. Note that when $P[i]$ is an opening parenthesis $excess(i)$ is the depth of the corresponding node, while in case of a closing parenthesis, it is the depth minus 1. Then, the main core parentheses operations can be defined as:

- $findclose(i)$ returns the position j of the closing parenthesis matching the opening parenthesis at $P[i]$: $min_{j>i}\{j \mid excess(j) = excess(i) - 1\}$.

- $findopen(i)$ returns the position j of the opening parenthesis matching the closing parenthesis at $P[i]$: $max_{j<i}\{j \mid excess(j) = excess(i) + 1\}$.

- $enclose(i)$ returns the position j of the opening parenthesis enclosing the opening parenthesis at $P[i]$[15]: $max_{j<i}\{j \mid excess(j) = excess(i) - 1\}$.

[15]That is, this operation gives the position of the opening parenthesis corresponding to the parent of a node.

Now, let us consider $excess(i,j) = excess(j) - excess(i-1)$[16]. Two primitive operations constitute the kernel of the FF approach:

- $fwd\_search(i,d)$ returns the smallest $j > i$ such that $excess(i,j) = excess(j) - excess(i-1) = d$.
- $bwd\_search(i,d)$ returns the greatest $j < i$ such that $excess(j,i) = excess(i) - excess(j-1) = d$.

These operations can be used to express the aforementioned core parenthesis operations (base of the basic tree operations like, for instance, *parent*, *subtreesize*, *nextsibling*, or *prevsibling* [MR01]), together with other sophisticated tree operations:

$$\begin{aligned}
findclose(i) &\equiv fwd_search(i,0) \\
findopen(i) &\equiv bwd_search(i,0) \\
enclose(i) &\equiv bwd_search(i,2) \\
level_ancestor(i,d) &\equiv bwd_search(i,d+1) \\
level_next(i) &\equiv fwd_search(findclose(i),0) \\
level_prev(i) &\equiv findopen(bwd_search(i,0))
\end{aligned}$$

Hence, the efficiency of FF stems from its ability to compute fwd_search and bwd_search in constant time thanks to the *range min-max tree*. This data structure is built over the (virtual) array of $excess(i)$ values as follows. The sequence P is split into blocks of size $s = \frac{w}{2}$ [17]. Then, for each block, the minimum and maximum excess values within the block are stored. After that, blocks are recursively assembled into groups of size $k = O(w/log\ w)$, in such a way that each new formed *superblock* stores the minimum and maximum excess within the blocks it holds. That results into a k-ary balanced search tree, the so-called *range min-max tree*. The total amount of space used is $O(n\ log(s)/s) = o(n)$ bits. In Figure 3.15 we show an example of *range min-max tree*, where $s = k = 3$.

To compute $fwd\_search(i,d)$ by using the *range min-max tree*, we first check if the answer is in the block i belongs to. Let us consider that this block, $q = \lfloor i/s \rfloor$ corresponds to range $[l_q, r_q]$ of P. The block scanning is done in constant time, with table lookups over a simple precomputed table[18]. If unsuccessful, the range $[r_q+1, n-1]$ of P, represented by range min-max tree nodes, is then examined.

[16]Notice that $|excess(i) - excess(i-1)| = 1$ for all i. In case $P[i]$ is an opening parenthesis, then $excess(i) - excess(i-1) = 1$. If $P[i]$ is a closing parenthesis, then the same subtraction results into -1.

[17]Remember that w is the machine word length and that $w \geq log\ n$.

[18]This table stores for all the different s-bit streams that constitute the different blocks of size s in P, the position where a target excess occurs.

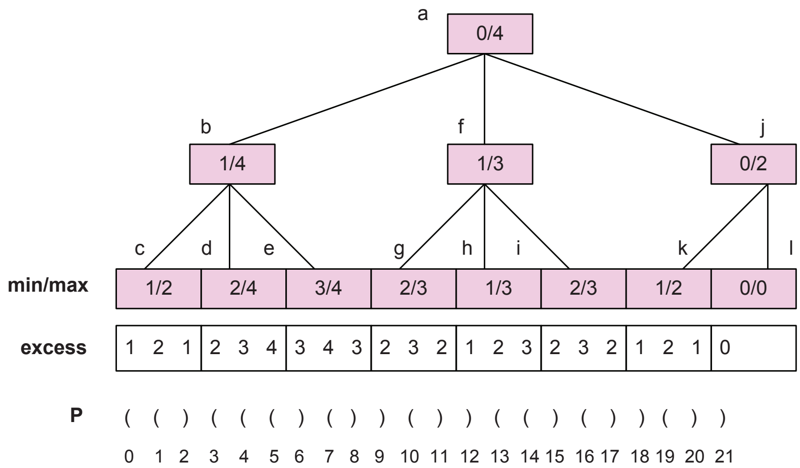


Figure 3.15: An example of the *range min-max* tree.

For each node, we verify if its minimum/maximum excess range, translated into absolute, contains $excess(i-1)+d$. Once the proper range min-max tree node is found, we know that the answer to $fwd\_search(i,d)$ lies within it. If it corresponds to an internal node, we iteratively go down finding the leftmost child that contains the desired excess[19], until reaching a leaf block, which will be finally scanned to find the exact value by table lookups, as before. An analogous procedure will be performed to compute $bwd\_search(i,d)$.

For instance, let us compute $findclose(3) = fwd\_search(3,0)$ in the example of Figure 3.15. Notice that it is equivalent to find the first $j > 3$ such that $excess(j) = excess(3-1)+0 = excess(2) = 1$. Therefore, we start by examining the node $\lfloor 3/s \rfloor$, that is, the node d in Figure 3.15. Since the target value 1 is not in that block, we continue the process by checking the minimum/maximum values of the nodes that cover the range $[5\ldots21]$, which turn out to be that corresponding to nodes e ($[6\ldots8]$), f ($[9\ldots17]$), and j ($[18\ldots21]$). In this way, we next scan node e. Again e does not contain the answer either, so we examine node f. Because $1 \le 1 \le 3$, that is, the minimum and maximum values of f enclose the target value, the answer must exist in its subtree. Therefore we explore the children of f from left to right, and find the leftmost one that contains the target value. In this case, it is node h. Given that it is already a leaf, we just scan its content using a precomputed table, and obtain that the answer to $findclose(3)$ is 12.

[19]Again, it is done in constant time, by using a precomputed table that provides for all the patterns of k/c (c being a constant) minimum/maximum values stored in the children of a node of the range min-max tree, the first child of the node whose minimum and maximum values enclose the target value.

Chapter 4

XML Storage and Querying - State of the Art Revision

Since their introduction, the growing interest and challenge of XML query languages has triggered much research to provide efficient solutions either as theoretical proposals or in the form of real systems. Likewise, in line with the development of systems focused on query aspects, several works have addressed the space challenge that the verbosity of XML documents entails, in the form of XML compression techniques. Many of these methods also tried to keep some kind of query support, leading to the so-called queriable compression tools.

In this chapter, we make a complete revision and look through some of the most relevant solutions from both areas. Section 4.1 first presents some well-known systems specifically designed to provide XML query support, either as streaming approaches (Section 4.1.1) or based on indexed proposals (Section 4.1.2). In turn, Section 4.2 focuses on XML compression, and starts by introducing a classification of XML compressors in Section 4.2.1. Then, Sections 4.2.2 and 4.2.3 close the chapter by providing a detailed description of the most important queriable and non-queriable XML compression tools.

4.1 XPath Query Systems

Regarding the XPath query language, typical query systems are usually divided into two different categories: those that follow a *streaming* approach (such as XSQ [PC05], SPEX [Olt07] and GCX [SSK07]), hence having to sequentially read the document to answer each query; and the *indexed* ones (such as Galax [FSC+03], Saxon [Kay08], Qizx/DB [Qiz], MonetDB/XQuery [BGvK+06], etc.), requiring a first preprocessing of the document to build additional data structures over it, that are then used to solve the queries without sequentially traversing the whole document. *Indexed* approaches can be further categorized into *in-memory* engines and *database* systems. Next, we describe some of the most representative examples from each category.

4.1.1 Sequential Solutions

Sequential solutions aim to be as close as possible to just performing one pass over the data, while keeping little main memory consumption to hold intermediate results and data structures. Within the sequential proposals, the three following engines constitute some well-known state of the art solutions, each of which provides different levels of query support:

XSQ. This engine [PC05, XSQ] addresses the problem of evaluating XPath queries over streaming XML. It supports queries limited to *child* and *descendant* axes, and predicates with at most one step. The idea behind this query engine is to use a hierarchical arrangement of pushdown transducers (HPDT) augmented with queues for buffering.

Automaton-based methods are commonly used for processing streaming data. Simple and linear XPath queries without predicates can be transformed into finite state automata that immediately output the relevant parts of the data, as soon as they are encountered [GMOS02]. However, when predicates, closures and aggregations are present in the query, its evaluation may become challenging, since when the automaton encounters a potential result, the data required to determine whether it must be or not in the final result may still have not been processed. For instance, if we consider the query `/journal[./year=2000]/title`, it may occur that the `year` child of a `journal` element appears after its `title` child. Hence, only when the first one is encountered, we can decide if the processed `title` element should be sent to the output or not. XSQ faces those challenges, by using pushdown transducers together with queues to buffer potential result items. A pushdown transducer (PDT) [Gur89] is a pushdown automaton (PDA), a variation of a finite state automaton that makes use of a stack [HU79], with an additional output tape. At each step, given the current state, a new symbol from the input tape and the symbol of the stack, the PDT changes to a new state and manipulates the stack according to the transition function. Moreover, an output can be generated during transition if the corresponding output operation is defined in the transition function.

PDTs used by XSQ, called *Basic PDT* (BPDT), differ from the originals, in that they are augmented with a buffer organized as a queue. In this way, output operations in BPDTs can also be buffering operations such as *enqueue(v)* (to introduce a specific item, v, into the queue), *clear()* (to remove all items from the queue), *flush()* (to send all items in the queue to the output), and *upload()* (to move all the items in the queue to the end of the queue of its

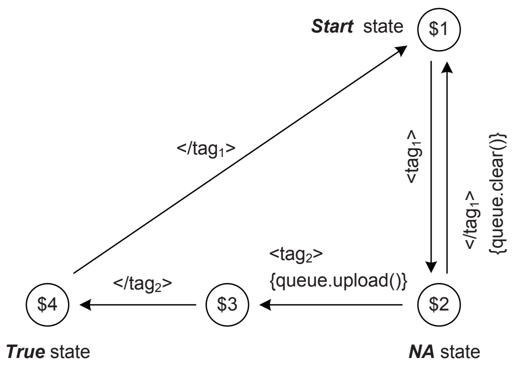


Figure 4.1: BPDT template for `/tag`$_1$`[./tag`$_2$`]`.

BPDT parent). XSQ defines a BPDT template for each different category in which the location steps, of the XPath queries that it considers, can be classified based on the items upon which predicates are evaluated (e.g. `/journal[ref]`, `/journal[ref="AF43"]`, `/journal[./title]`, `/journal[./title="ACM TODS"]`, `/journal[./title/id="TF25"]`, etc.). In Figure 4.1 an example of BPDT template for a location step matching `/tag`$_1$`[./tag`$_2$`]` is shown. Notice that every BPDT always has a *true* state, to indicate that the predicate has been evaluated to true, and a NA state, that indicates that the predicate has not yet been evaluated. Returned transitions from the NA state to the *start* state, means that the predicate has not been fulfilled. The logic of the predicate is encoded in the BPDT. Therefore, given a complex query, each of its location steps is represented by a BPDT, which are further combined into a hierarchical pushdown transducer (HPDT), in the form of a binary tree, encoding the complete query. Depending on their position inside the HPDT arrangement, BPDTs can determine whether a predicate has been already evaluated or not and hence buffer operations are also settled accordingly. For instance, when creating a HPDT, *upload()* operations of generic BPDT templates may be replaced by *flush()* ones, if a BPDT is related with its BPDT parent through the *true* state. If not, potential results, must be enqueued in the parent until it validates the predicate.

SPEX. SPEX [Olt07, SPE] is another example of query processor that evaluates XPath queries over XML data streams. Like XSQ, SPEX uses pushdown transducers (PDT) to perform query evaluation, however it does not need additional buffers. The query language supported by SPEX is the forward core of XPath [GKP05], extended with path union and path difference. Prior to evaluating a query, SPEX rewrites the query into an equivalent one, without reverse axes [OMFB02].

Then, a network of simplified pushdown transducers is created by materializing each different query component into a single-state deterministic pushdown transducer. The transducers use their stacks to model partial matchings, and their tapes, to communicate with other transducers. That is, the output tape of a transducer T_i becomes the input of the transducer T_{i+i}. These inputs and outputs are basically annotations used to mark selected nodes during evaluation and to also record predicates satisfaction. SPEX also uses specialized transducers, called *filter* transducers, to minimize the stream fragment processed by transducers in a network, in such a way that only relevant input fragments for the correct evaluation of the query are sent from an arbitrary transducer to its successors.

GCX. Unlike the previous streaming processors, GCX [SSK07] is an engine that supports XQuery evaluation (besides XPath). However, its most relevant feature is that, to keep main memory consumption low, GCX uses a buffer management scheme that combines static and dynamic analysis to effectively purge main memory buffers based on the progress in query evaluation.

a)

Query:

```
<result> {
  for $s in /store return
    ((for $p in /store/* return
      if (not(exists $p/price)) then $p else ()),
    for $x in /store/product return  $x/name)
} </result>
```

b)

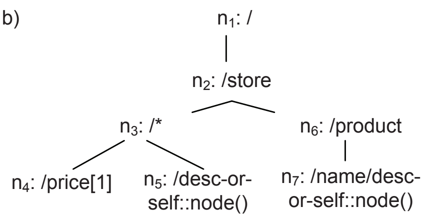


Figure 4.2: An XQuery expression (a) and its corresponding projection tree (b).

```
<result> {
  for $s in /store return
    ((for $p in /store/* return
        (if (not(exists $p/price)) then $p else (),
        signOff($p, r3), signOff($p/price[1], r4),
        signOff($p/desc-or-self::node(), r5))),
      (for $x in /store/product return
        ($x/name,
        signOff($x, r6),
        signOff($x/name/desc-or-self::node(), r7))),
      signOff($p, r2))
} </result>
```

Figure 4.3: Query rewritten with *signOff* statements.

GCX extends the static *document projection* technique [BCCN06, MS03]. Given a query, the static analysis of GCX derives its *projection tree*, that is, the parts of the input document that should be buffered. Each projection tree node, n_i, defines a role r_i. For example, given the XQuery query of Figure 4.2 a), its derived projection tree is shown in Figure 4.2 b). Notice that it only seeks relevant fragments for query evaluation. Hence, while parsing the input XML stream, a projected version will be computed, buffering only data that is relevant to query evaluation, and discarding the rest. Those buffered tokens will be assigned the corresponding role on-the-fly. In addition, the query evaluation moments in which buffered nodes lose their roles are determined at compile-time. To this aim, a query rewritten is performed, by inserting *signOff* statements that indicates which nodes become irrelevant at that point for the remaining query evaluation. Then, at run-time, the buffer manager is notified to update the roles of buffered nodes, when these statements are encountered. Once a node loses all its roles, it can be safely deleted if none of its descendants is assigned any role, thus cleaning buffers dynamically. In Figure 4.3, we show an example of query rewritten, corresponding to the query of Figure 4.2 a). This global buffer management scheme is called *active garbage collection*.

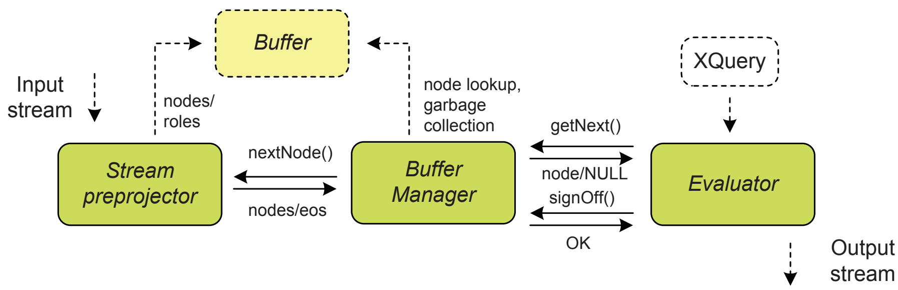


Figure 4.4: GCX global architecture.

The GCX architecture composed of a *stream preprojector*, a *buffer manager* and the *query evaluator* (see Figure 4.4), performs a query evaluation according to the aforementioned scheme in a *pull-based* manner. After having extracted the query projection and rewritten the query, the query evaluator starts by evaluating the query until it has to *block* either because a new node is required or a *signOff* statement is reached. In both cases, the buffer manager is invoked, and query evaluator remains blocked until obtaining an answer. In case new data that is not buffered is requested, the buffer manager calls the stream preprojector to consume data from the input stream by matching tokens against the projection tree, until it encounters relevant data. Matched tokens are then copied into the buffer, together with the corresponding roles, and later handled by the buffer manager in its communication with the query evaluator. In turn, if buffer manager receives a *signOff* statement, it triggers the *active garbage collector*, to make nodes lose specific roles and even to delete some of them that may become irrelevant at that stage of the query evaluation.

Step	Input stream	Buffer contents	Output stream
1			<result>
2	<store>	store $\{r_2\}$	
3	<product>	store $\{r_2\}$ | product $\{r_3, r_5, r_6\}$	
4	<name/>	store $\{r_2\}$ | product $\{r_3, r_5, r_6\}$ | name $\{r_5, r_7\}$	
5	<model/>	store $\{r_2\}$ | product $\{r_3, r_5, r_6\}$ name $\{r_5, r_7\}$ model $\{r_5\}$	
6	</product>	store $\{r_2\}$ | product $\{\cancel{r_3}, \cancel{r_5}, r_6\}$ name $\{\cancel{r_5}, r_7\}$ model $\cancel{\{r_5\}}$	<product> <name/> <model/> </product>
7		store $\{r_2\}$ | product $\{r_6\}$ | name $\{r_7\}$	

Figure 4.5: Example of active garbage collection in GCX query evaluation.

Figure 4.5 illustrates some steps of the evaluation of the query of Figure 4.3, with respect to a sample input stream `<store><product> <name/> <model/> </product>....` At each step, the input stream, as well as the buffer contents and the generated output, are shown. In step 1, the start-tag `<result>` is output. Next, the query evaluator enters the first *for* clause. Given that no data are still available in the buffer, the evaluator remains blocked. At step 2 `<store>` is read. It matches n_2 projection, thus it is inserted into the buffer together with the role r_2.

Then, the query evaluator binds `$s` with the just copied `<store>` node. After that, it tries to execute the second *for* clause, but again, there is no relevant data in the buffer. Therefore, in step 3 a new token from the input stream is read, `<product>`, that matches several roles, namely r_3, r_5 and r_6, and it is associated with variable `$p`. Yet it is not possible to evaluate the subsequent *if* condition, hence the input stream is processed once more. In step 4, `<name/>` is copied into the buffer with the roles r_5 and r_7. However it is still not possible to perform the *if* condition evaluation, so `<model/>` is also processed from the stream and buffered with role r_5 in step 5. At this point, the *if* expression keeps blocked, thus step 6 reads a new token, `</product>`. Having encountered the end of the node bound to `$x`, the *if* clause is evaluated, leading to the output of the bound node. Next the evaluator finds a sequence of *signOff* statements. These are then sent to the buffer manager, which updates the roles of the buffered nodes accordingly, and also deletes those that are not relevant for the query evaluation any more (see removed `model` node in step 7). Query evaluator would continue evaluation of the following *for* clause in a similar way.

4.1.2 Indexed Solutions

Unlike sequential solutions, indexed ones prioritize the efficiency in query evaluation through the use of indexes that avoid to sequentially scan the XML input document at each run. However, their main drawback arises from the fact that indexes may incur into high space requirements. In general, indexed approaches can be classified as in-memory processors or database systems, depending on a persistent storage of the data is or not provided.

4.1.2.1 In-memory Engines

These solutions do not provide a persistent storage. They usually use machine pointers to represent XML data into main memory, which tends to blow up memory consumption. Two well established processors are:

Galax. Galax [FSC+03, Gal] is a main-memory processor supporting XPath, XQuery, and some extensions for XML updates and scripting[1]. Its architecture comprises three main modules, each one related to XML documents, XML Schema, and XQuery processing, respectively. The input XML document is parsed by the first module in a streamed fashion using SAX, and loaded into memory as an XML data model instance [DOM, XDM]. This data model provides the necessary information for further query processing, keeping for each node (i.e. document, element, attribute, and text), accessors that return its name, base URI, type, typed

[1] There is an implementation of Galax supporting the *full text* extension of XQuery [Ful] called GalaTex [CAYBF05].

value, unique node identifier, or global document order, as well as pointers to parent nodes, children, etc. On the other hand, the XQuery module is in charge of the query parsing and evaluation plan production. Given a query, this module first creates the abstract syntax tree representation (AST) of the query, and after some normalization and optimization operations, transforms it into an evaluation plan in Galax algebra [RSF06]. This plan is then applied over the data model representation of the input document. The XML Schema module, in turn, is used by the two previous modules to validate the input XML document, and to perform query static typing, respectively, whenever documents have associated XML Schemas.

Saxon-HE. This is the open source version of the well-know Saxon processor [Kay08, Saxb], that provides implementations of XPath, XQuery and XSLT at the basic level of conformance. Like Galax, being a main-memory query engine, Saxon creates for the input XML document an in-memory tree representation. However in that case, it offers two different implementations proprietary to Saxon, a typical linked tree, where an object is created for each node (e.g. DOM model), and another one inspired by the DTM model of Xalan [Xal], that makes use of integer arrays and pools of strings to represent the structure and content of the XML document.

4.1.2.2 Database Systems

Database systems provide a persistent storage. Indexes are initially loaded into main memory the first time data is processed. Yet, an important shortcoming is that, in case indexes require much space, they may be manipulated on disk. This feature implies usually high I/O transfer times that may seriously affect the overall efficiency in query processing.

Within this category, we can find native XML databases, but also relational ones. Three of the most representative systems are next presented:

eXist. It is an open source native XML database system [Mei02, EXI]. eXist provides schema-less storage of XML documents in hierarchical collections and index-based query processing of XPath and XQuery, also including their full text extensions, as well as XSLT and XUpdate [XUp] support.

Four different index files constitute the core of the storage back-end of eXist (see Figure 4.6). All of them are based on $B+$ trees:

- *collections.dbx*: this index file manages the collection hierarchy and maps collection names to collection objects. An unique identifier is assigned to each collection and document during indexing.
- *dom.dbx*: it is the backbone of eXist, and consists of a single paged file in which all document nodes are stored according to the DOM model. To

uniquely identify each node, eXist uses a pair $\langle docId, nodeId \rangle$, being the first component the identifier of the document it belongs to, and the second one, a numbering scheme that allows to directly determine node relationships, thus avoiding to keep track of links between nodes. This numbering scheme corresponds to the *Dynamic Level Numbering* (DLN) [BR04], inspired by Dewey's decimal classification [TVB$^+$02]. Conceptually, the identifier of a node is composed of a sequence of numeric values separated by a dot. The root node is assigned a single numeric value. Each child node identifier starts with the node identifier of its parent appended by a dot and a numeric value called the *level value*. That is, sample identifiers could be 1, 1.1, 1.1.1, 1.2, etc. These identifiers are further encoded using for each level value a variable-length encoding of fixed size units of 4 bits.

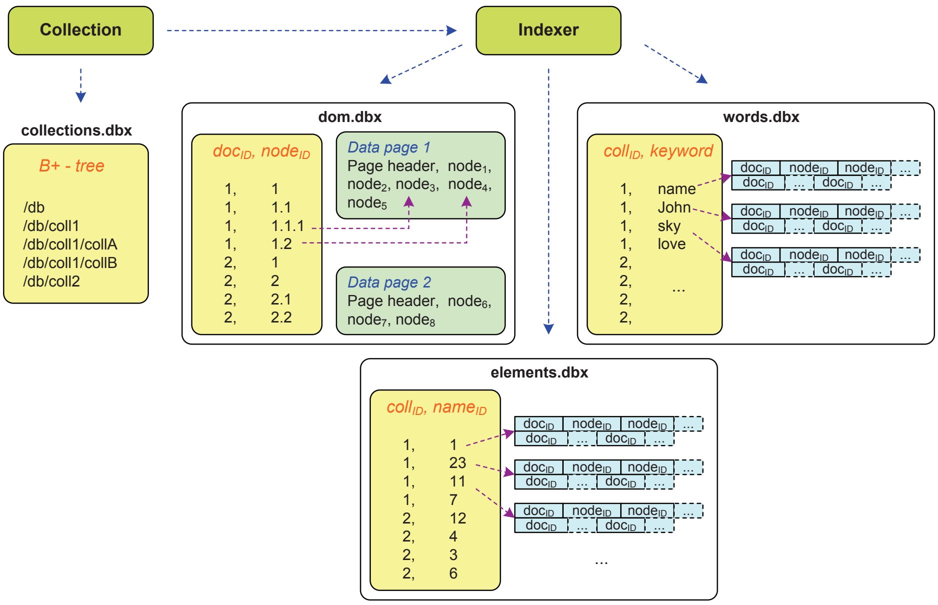


Figure 4.6: Storage architecture of eXist.

- *elements.dbx*: for each element and attribute, eXist creates an entry in this file. Hence, given a pair $\langle collId, nameId \rangle$, eXist stores an ordered list of documents and node identifiers, where the qualified name, *nameId*, appears. Since the sequence of document and node identifiers consists of integer values, a combination of delta and variable byte codings are used to save storage space.

- *words.dbx*: similarly to the previous index, this file maps extracted keywords from text nodes and attribute values, to their corresponding documents and node identifiers.

Since the access to the persistent DOM representation is always expensive, eXist tries to process queries avoiding to load and traversing the actual DOM nodes, based on its indexing scheme. For instance, most of the structural-based queries are solved using path join algorithms.

Qizx/DB. Qizx/DB is also a native XML database system [Qiz] fully supporting XQuery, and its full text extension. It also provides support for XUpdate and XSLT, nevertheless it has been optimized for high querying speed, rather than for intensive updating of XML data. Qizx/DB creates and exploits the following indexes:

- *Elements index*: provides direct access to elements by name. It also contains information about structural relationships like *child* or *descendant*.
- *Attributes index*: Qizx/DB distinguishes three different attribute indexes according to the type of the attribute value: text, numeric or date.
- *Simple elements content*: given an element and a value, this index returns all elements that enclose a *simple content* (that is, a sequence of characters without whitespaces) corresponding to the value. As done with respect to the attribute index, simple contents are also indexed depending on their type.
- *Full text index*: a word-based index for elements data content.

Documents and indexes are compressed, to reduce disk space use and I/O transfer time.

MonetDB/XQuery. Unlike eXist and Qizx/DB, MonetDB/XQuery [BGvK+06] is a relational database management system providing full support of XQuery and XUpdate. It also supports some full-text capabilities through the use of the PF/Tijah text index [LMR+05].

MonetDB/XQuery basically consists of the *Pathfinder* XQuery compiler [GST04], on top of the MonetDB RDBMS [Bon02]. Pathfinder assumes XML documents transformed into a relational encoding that maps each node v in the document tree onto a two-dimensional plane, given by its preorder and postorder rank. In particular, this information is encoded by representing each node with a 3-tuple $\langle pre(v), size(v), level(v)\rangle$, recording the preorder rank of v, the number of nodes in the subtree below v, and the distance of v from the root[2]. Further tables are also maintained by the system to store additional node properties (e.g. kind of

[2]Note that the postorder rank of v can be computer as $post(v) = prev(v) + size(v) - level(v)$.

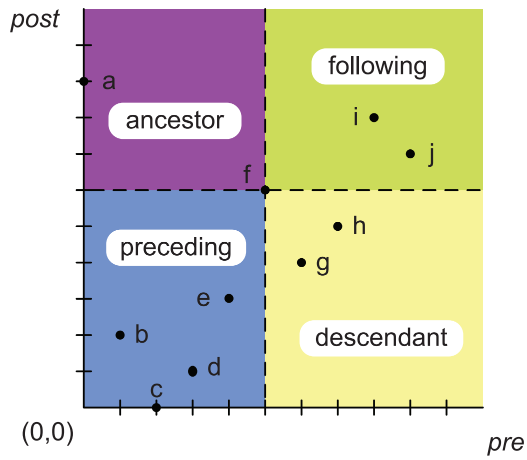


Figure 4.7: XPath axes correspondence in the *pre/post* plane for the context node f.

node, qualified name, textual content, etc.). However, the relevant feature is that this encoding scheme efficiently characterize XPath axes as regions in the *pre/post* plane (see Figure 4.7), thus turning their evaluation into a relational range selection in that plane, powered by index structures (e.g. B-trees) [GvKT04].

Under this scheme, every incoming XQuery expression is compiled by Pathfinder into a purely relational query plan, that operates on the aforementioned tree encoding. Yet to improve XPath processing, *tree-awareness* is also introduced into the relational query evaluator, by means of the *staircase join* [GvKT03], that extends the relational join operator. Taking into account that a step is generally evaluated on a sequence of context nodes, the *staircase join* introduces three main tree-aware optimizations into the join operator: *i*) pruning, *ii*) partitioning, and *iii*) skipping. The two former avoid duplicating results generation. *Pruning* stands for omitting those context nodes that are *included* into the quadrant covered by another context node. An example of this technique is shown in Figure 4.8 a). In turn, *partitioning* tries to cope with partial overlaps, by partitioning the regions along the *pre* axis. In Figure 4.8 b), this technique is applied to the sequence of context nodes obtained after applying the *pruning* optimization of Figure 4.8 a). Finally, the *skipping* strategy, aims to avoid unnecessary nodes processing, as depicted in Figure 4.8 c). The same strategies are also adapted to work with XQuery expressions, leading to the so-called *loop-lifted staircase join*.

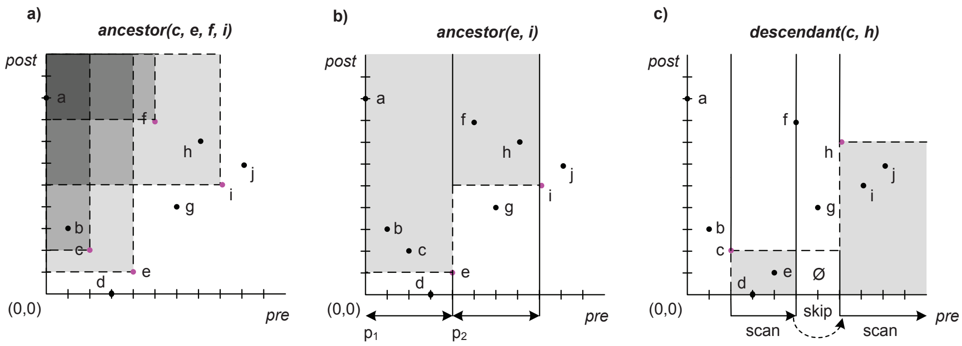


Figure 4.8: *a*): context nodes c and f are *pruned*, since they are inside the *ancestor* region of e and i. *b*): the overlapping *ancestor* regions covered by e and i are partitioned along the *pre* axis at *p1* and *p2*. *c*): after hitting f, *descendant* staircase join infers that no results can occur until h, thus a large part of the *pre/post* plane is skipped.

4.2 XML Compression

As stated in previous sections, space may result into a key factor. Indeed, another quite active line of research in the last years has been XML compression. Compression has been acknowledged to save space, which may be decisive to avoid using secondary storage, to use fewer machines, or even to achieve a feasible solution when the memory is limited (as in mobile devices). However, it also saves time. Time is the critical factor in efficiency, and processing a compressed version of a document saves time when it is transmitted through a network, when we need to access to disk for a document, or more importantly, when it is processed. Therefore, compression is clearly more convenient.

With regards to the XML context, where query languages are so relevant, many of the proposals developed have also considered query aspects rather than just focusing on space savings. These works, known as XML queriable compression tools, try to keep little space requirements, while providing some kind of query support. Some of them allow performing queries directly over the compressed representation of the text (either sequentially or using indexes), while some other have to fully/partially decompress the data before querying them. Although these tools constitute the most interesting approaches within the scope of this thesis, today there is a stated lack of available practical solutions [Sak09].

Following sections include a complete review of the XML compression methods that have been recently proposed regardless their query abilities, as some of the non-queriable proposals will be later referred in Chapter 9 for an in-deep evaluation regarding their compression properties. An initial classification of XML compressors is presented in Section 4.2.1. Then, Section 4.2.2 and Section 4.2.3 are devoted to describe some well-known tools of each category.

4.2.1 Classification of XML Compressors

XML compression can be seen as a particular field of text compression, which deals with semi-structured documents. Indeed, most times XML documents are treated as text files, and hence general purpose text compressors are used to compress them. This feature leads to a first classification of XML compressors, depending on their awareness of the XML documents structure. Thus, according to this, XML compression techniques are separated into two main categories:

- **General text compressors** : also called as *XML-blind* compressors, these compressors treat XML documents as plain text files and do not care about their structure. Traditional text compressors such as those mentioned in Section 3.1.3 fall into this group.
- **XML conscious compressors** : these compression techniques are aware of XML documents structure and exploit this knowledge to achieve better compression ratios than the compressors of the previous group. XML conscious compressors can be further divided into:
  - *Schema dependent compressors*: the compression method requires the associated schema information of an XML document to be accessed.
  - *Schema independent compressors*: compression can be performed without the XML document schema being accessed.

  Although the first ones are intended to obtain higher compression ratios, the necessity of working with the XML documents schema information, which is commonly not available, rather restricts their use in practice. Examples of compressors of the first category are Millau [GS00, SM02], SCA [LW02], XAUST [SS05], and RNGzip [LE07], while XMill [LS00], XMLPPM [Che01], SCM [ANF07], and Exalt [Tom03], as well as, XGrind [TH02], XPRESS [MPC03], XCQ [LNWL03, NLWL06], XQzip [CN04], and XBzipIndex [FLMM05, FLMM09] are other typical methods of the second class.

Moreover, it is possible to devise a second classification of XML compressors with respect to their query support.

- **Non-queriable compressors** : these XML compression techniques do not allow any kind of query evaluation. Instead, they aim to achieve the highest compression ratio. All general text compressors belong to this category. However, we also can find non-queriable XML conscious compressors, such as Millau [GS00, SM02], XAUST [SS05], XMill [LS00], SCM [ANF07], XComp [Li03], Exalt [Tom03], AXECHOP [LDM05], etc.

- **Queriable compressors** : both compression and querying are important aspects for these techniques, that usually compromise compression ratio for sake of query processing. Their main focus is to allow query evaluation without full text decompression, just requiring either a partial decompression (e.g. QXT [SGS08], XCQ [LNWL03, NLWL06], XQzip [CN04], XMLZip [XMLb], etc.) or, ideally, being able to process queries directly over the compressed XML document (e.g. XGrind [TH02], XPRESS [MPC03], XQueC [ABMP07], XSeq [LZLY05], XCPaqs [WLLH04], ISX [WLS07], TREECHOP [LMD05], LZCS [ANF07], XBzipIndex [FLMM05, FLMM09], etc.). By default, all queriable XML compressors are XML conscious compressors as well.

  Queriable XML compressors can be further classified into *homomorphic* and *non-homomorphic* compressors. We know as *homomorphic* compressors those techniques, such as XGrind [TH02], XPRESS [MPC03] and QXT [SGS08], that preserve XML document *conformation*. That is, they do not separate structural and data parts when compressing an XML document, unlike *non-homomorphic* compressors do (e.g. XCQ [LNWL03, NLWL06], XQzip [CN04], XMLZip [XMLb], XQueC [ABMP07], XSeq [LZLY05], XCPaqs [WLLH04], ISX [WLS07], TREECHOP [LMD05], LZCS [ANF07], XBzipIndex [FLMM05, FLMM09], SXSI [ACM$^+$10], etc.). Therefore, homomorphic compressors allow one to access/index the compressed version in a similar manner as when working with the original XML document, since the former can be seen as the result of a simple mapping/replacemnt.

Since general purpose text compressors have been previously discussed in Sections 3.1.4, 3.1.5, and 3.1.6, we next focus on XML conscious compression techniques, and present some well-known examples of non-queriable and queriable compressors, according to the classifications mentioned above. Figure 4.9 shows an scheme summarizing the tools that will be further described.

4.2.2 Non-Queriable XML Compressors

As non-queriable XML compressors aim to get outstanding compression ratios, careless of providing any kind of query support, they may use XML documents schema information to improve compression. Hence, these methods are usually presented by considering a division regarding this feature.

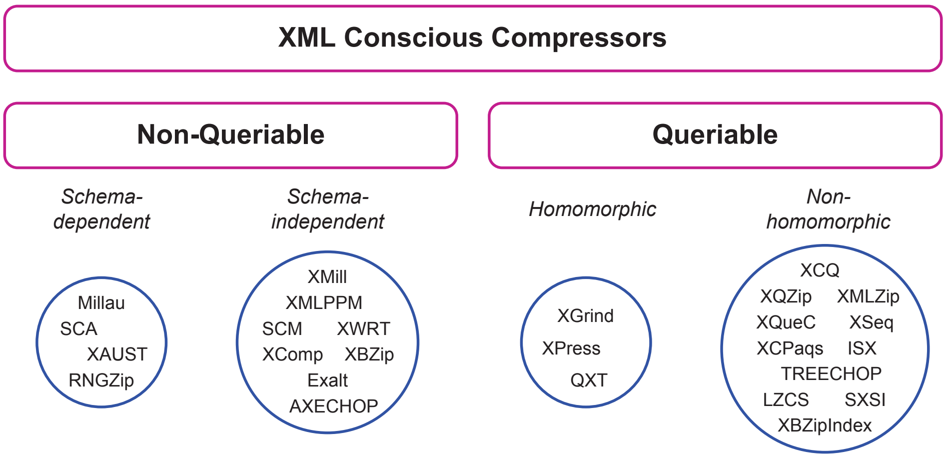


Figure 4.9: Classification of some examples of XML compression tools.

4.2.2.1 Schema Dependent Compressors

Those compressors make use of an XML document DTD to perform compression/decompression. Among this kind of compressors, Millau, SCA, XAUST and RNGzip are four of the most important tools.

Millau. The WBXML (Wireless Application Protocol Binary XML) Content Format Specification defines a compact binary representation of XML, to reduce transmission size of XML documents without loss of functionality or semantic information. Yet this encoding format only considers tags and attribute names, it does not compress at all character data content nor attribute values. Millau [GS00, SM02] follows the essence of WBXML, but it extends it with separation of structure and content, improving the compression algorithm itself. While compressing, Millau generates separated streams. The structural stream is encoded by using the WBXML encoding. In turn, the content stream is compressed by using general text compression techniques like *deflate* [Deu96]. In addition to structure and text division for compression, Millau can also take advantage of the Document Type Definition (DTD) of an XML document, and optimize structural compression. In that case, the applied technique, called *Differential DTD Tree Compression* (DDT), only encodes the differences between the schema and the document. That is, minimal structure information is stored, since only the occurrences of DTD operators such as ? (optional operator), | (decision operator), and $+$ and $*$ (repetition operators), need to be encoded, yielding to an efficient storage. Moreover, Millau may perform content grouping to improve data part compression.

SCA. Similarly to Millau, SCA [LW02] also uses DTD information to enhance the compression of the document structure by only encoding the information that can not be inferred from the given DTD, and extracts content part to a separate container to be then compressed by a generic compressor such as *gzip*. However, the main difference with respect to Millau lies on the followed approach to process both inputs, the DTD and the XML document. Whereas Millau simultaneously parses the DTD tree and the DOM tree of the XML document, generating structural and content streams, SCA first creates a special tree by combining the DTD information and the XML document. This tree is then processed by a *pruning* phase leading to a reduced version of it. The tree first created is essentially a DOM representation of the XML document, but with added DTD operator nodes such as $*$, $|$, ?, etc. The pruning step is in charge of reducing this tree by only keeping those nodes that are necessary to infer the correct structure (i.e. those that can not be derived from DTD any other way), drawing as well data values to a separated content stream to be further compressed. In a final step the reduced tree is traversed and encoded following a *breath-first* (BFS) order.

XAUST. XAUST [SS05] is an on-line compression scheme that tries to exploit the knowledge encapsulated in a DTD specification by means of a set of deterministic finite automata (DFA), one for each element, directly generated from the DTD of the document. Using this information, XAUST is able to track the document structure, and to make accurate predictions of the expected symbols. Transitions of each automaton are labeled by element names, while states can have a single output transition, or more than one. In the first case, no symbol encoding needs to be performed. Only when multiple outgoing transitions are possible, the element labeling the transition is encoded using an arithmetic encoder for the state. Whenever a transition is taken, scheme transits to the start state of the DFA corresponding to the element in the label. Regarding to character data and attributes, every element that may enclose some of these items, will have an associated container which is incrementally compressed using a single model for an arithmetic order-4 compressor [WNC87].

RNGzip. RNGzip [LE07] also applies the idea of not transmitting information that is already known, but in this case, from the RELAX NG schema [CM01] instead from the document DTD. RNGzip builds a deterministic tree automaton from a specified schema, and given an XML document it only needs to produce symbols whenever a *choice point* or a *text* transition is encountered. In the former situation, RNGzip transmits the transition taken, while in the latter, it sends the textual data. In both cases, the generated streams are then encoded by using distinct eligible compression schemes, namely *gzip*, LZMA, *bzip* and PPM.

4.2.2.2 Schema Independent Compressors

Unlike schema dependent compressors, those independent ones do not need the DTD additional information to compress/decompress an XML document. Consequently, they have experienced a widespread use along the years. Starting with XMill, which constitutes the first example of an XML conscious compressor, we will next review some of the most relevant proposals of this group (such as XMLPPM, SCM, XWRT, XComp, XBzip, Exalt and AXECHOP), each one based on different underlying schemes.

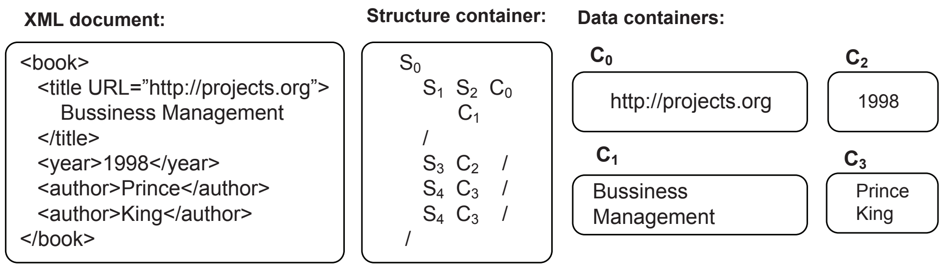


Figure 4.10: Example of text compression with XMill.

XMill. XMill constitutes the first approach to XML conscious compression. As its own authors state in [LS00], it is not by itself an actual compressor, but rather an *extensible* tool to specify and to apply different existing compression methods to compress XML data items. The main novel ideas behind XMill are to separate the *structure*, given by tags and attribute names, from the *data*, that is, element contents and attribute values; and to group data items into homogenous *containers*. Both structural part and data containers are then compressed separately. Regarding the structural items, XMill applies a *dictionary* based encoding scheme generating a compact representation where tag and attribute names are replaced by dictionary indexes, while data values are replaced with their container identifier. Figure 4.10 depicts an example of this structure representation for a sample XML document, where S_i represents dictionary codes given to start-tags and attribute names, /, is the token representation for end-tags, and C_i, encodes the container where each data value is stored. This final representation is then passed to a back-end general text compression scheme (usually *gzip*). On the other hand, data values are assigned to different data containers according not only to their data path, but also to data types. The aim is to group together items that are semantically related, by creating homogenous containers. For instance, in the example of Figure 4.10, years will be grouped in one container, while author names will be assigned a different one, and so on. The main reason behind this division is that some data items are text, others are dates, numbers, and even DNA sequences. Therefore, XMill applies specific and specialized compressors[3] (called *semantic compressors*) to each container, to get the best compression performance. Moreover, XMill allows the user to control the content of the data containers and the selection of semantic compressors as well, by means of *containers expressions* that are provided in the XMill command line. This may achieve further compression improvements than that obtained when using a default mode. However it claims for user expertise and effort to get the best compression. Finally, as happened with the structural part, all data containers are also compressed using a general compressor, commonly *gzip*, and concatenated in the output file.

The intended applications of XMill are data exchange and data archiving, to minimize network bandwidth consumption and to reduce space requirements, respectively. It has not been designed to support queries over the compressed text, so full decompression is needed before query evaluation.

XMLPPM. It is a streaming XML compressor [Che01] based on the Multiplexed Hierarchical Modeling (MHM) technique, that combines SAX encoding and the Prediction by Partial Matching compression scheme (PPM) [CW84]. The input XML document is parsed by a SAX parser generating a sequence of SAX events that are first encoded in binary format using a bytecode representation, called ESAX(*Encoded SAX*), and then processed by one of four PPM models depending on its syntactic context, namely elements and attributes names (*Syms*), elements structure (*Els*), attributes values (*Att*) and strings (*Chars*). That is, XMLPPM *multiplexes* different PPM models, to which encoded SAX events are sent according to their syntactic context, for running predictions and encodings. This provides benefits similar to those of XMill containers. Figure 4.11 shows an example of XMLPPM processing over an XML fragment. Notice that when an XML element is processed for the first time, its name string value is sent to the *Syms* model to be assigned a bytecode, since that element name has not been encoded before (e.g. 01 for `library`, 02 for `book`, and 03 for `title`). Then the given byte symbol is sent to the *Elts* model. Next times the same element is processed again, just the already assigned bytecode is passed to the *Elts* model. This same procedure is applied to attribute names (see `year` in Figure 4.11), but using *Atts* model instead of *Elts*. In turn, attribute and data values, are directly sent to *Atts* and *Chars* models, respectively, to be encoded. In the last case, also a special bytecode, FE, is sent to *Elts*. That is because ESAX encoding not only uses bytecodes to encode start-tags and end-tags, together with attribute names, but it also reserves particular bytecodes to indicate events like the beginning and end of character data, or even of comments. Observe that, for instance, all end-tags are replaced by FF.

Furthermore, to avoid breaking up dependencies of correlated symbols that hold into different syntactic classes and thus different PPM models, XMLPPM also

[3] XMill provides several built-in encoders that can be used, but it also allows one to link any other existing compresor. That is why XMill is defined as an *extensible* tool.

***Input* (1)**	**<library>**	**<book**	**year =**	**“2009”**	**>**	**<title>**	**The Universe**	**</title>**	**</book>**
Elts:	01	<01> 02				<02> 03	FE	FF	FF
Atts:			<02> 0A	2009 00	<02> FF				
Chars:							<03> The Universe 00		
Syms:	library 00	book 00	year 00			title 00			

***Input* (2)**	**<book**	**year =**	**“2009”**	**>**	**<title>**	**Amélie**	**</title>**	**</book>**	**</library>**
Elts:	<01> 02				<02> 03	FE	FF	FF	FF
Atts:		<02> 0A	2010 00	<02> FF					
Chars:						<03> Amélie 00			
Syms:									

Figure 4.11: Example of Multiplexed Hierarchical Modeling in XMLPPM.

injects previous symbols, regardless the model it belongs to, into the multiplexed models to be used as a context for a current symbol. The dependency between an element and its enclosed data is a common case of strong correlation, hence the enclosing element symbol is injected into the corresponding model before an element, an attribute or a data value is encoded (see bytecodes inside $\langle$ and $\rangle$ in the example of Figure 4.11). Those *injected* symbols indicate to the model that they have been seen but they are not explicitly encoded nor decoded, they only aim to retain dependencies. In [Che05], another variant of XMLPPM, named DTDPPM, that performs DTD-specific optimizations to compress XML documents regarding their DTD information was also presented. XMLPPM achieves, in general, better compression ratios than XMill, in its default mode. Yet its main drawback are compression times, since PPM compression family is known to be relatively slow.

SCM. In [ANF07], authors present the Structure Context Modeling (SCM) technique, whose main idea is to use different compression models to compress the text under each different XML tag (instead of considering complete *paths* from the root, like done by some of the previous compressors), and apply it into two variants, SCMHuff and SCMPPM, that use a Huffman coding and PPM modeling, respectively.

As a semistatic approach SCMHuff makes two passes over the text. In the first one, text is modeled by creating separated *dictionaries* (the set of vocabulary words together with the assigned codes) for each tag. Then, in a second pass, data under a specific tag are encoded according to the Huffman model obtained for that tag in the first step. Moreover, in that case, authors also consider the possibility of merging some of the models. To maintain separated dictionaries for each different tag may

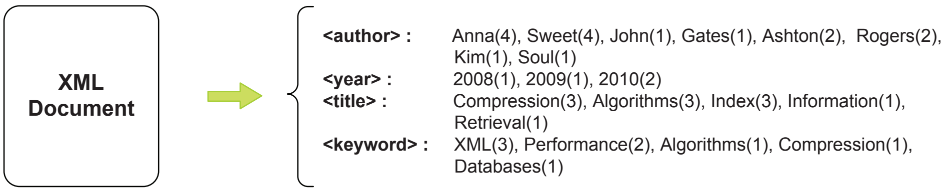


Figure 4.12: Dictionaries created from a sample XML document.

Table 4.1: Size contributions maintaining *i*) only one dictionary, *ii*) separated vocabularies for each tag, and *iii*) after merging *title* and *keyword* vocabularies.

	One dictionary				*Dictionary per tag*				$Dict_{title} \cup Dict_{keyword}$			
Dictionary	$V_d^\dagger$	n_d	H_d	T_d	$V_d^\dagger$	n_d	H_d	T_d	$V_d^\dagger$	n_d	H_d	T_d
author	144	39	3.965	299	64	16	2.750	108	64	16	2.750	108
year					24	4	1.500	30	24	4	1.500	108
title					40	11	2.163	64	64	19	2.800	118
keyword					40	8	2.156	58				

† Values computed assuming that we need 8 bits per different dictionary word, thus $V_d^\dagger = 8 * V_d$.

not pay off due to storage overhead. Therefore, if two dictionaries share most of the terms and have similar probability distributions, they are merged under a single one. To determine whether two dictionaries should be combined, without the need of running again Huffman algorithm over its union, authors propose a costless method based on the fact that Huffman compression is very close to the zero-order entropy of the text, and estimate the size of the resulting Huffman compressed text under a merge. To this aim, the estimated size contribution of a dictionary d, T_d, is computed by the following heuristic: $T_d = V_d + n_d * H_d$, where V_d is the size of the vocabulary that composes the dictionary, n_d is the total number of words, and H_d represents the estimated zero-order entropy of the dictionary, obtained by calculating terms vocabulary probabilities restricted to the specific dictionary scope. In this way, two dictionaries are merged if $T_i + T_j > T_{i \cup j}$, leading to a compression saving, $A_{i,j}$, given by $A_{i,j} = T_i + T_j - T_{i \cup j}$. For instance, let us consider the vocabularies associated to specific tags, namely `<author>`, `<year>`, `<title>`, and `<keyword>`, of an XML document sample depicted in Figure 4.12. Table 4.1 shows the benefits of using SCMHuff and dictionary merging advantages over that example.

The second variant of SCM is SCMPPM, that uses different PPM models for the text that lies under each different tag, hence it is considered as an extreme variant of XMLPPM. Since PPM is adaptive, there is no need to store models in the compressed file, and thus merge is not necessary either. SCMPPM achieves better compression ratios than SCMHuff, still unlike this one, SCMPPM does not provide random access nor direct search over the compressed document. The main flaw of SCMPPM are memory requirements to maintain multiple PPM models.

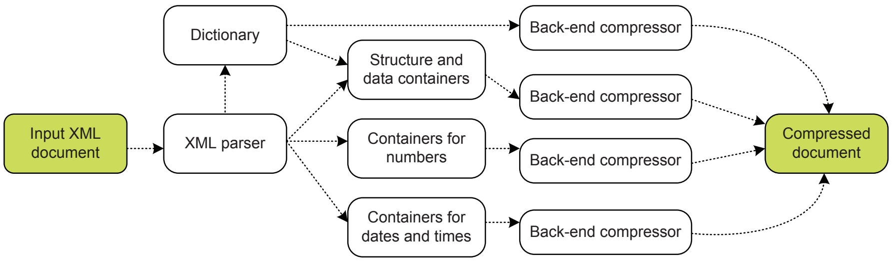


Figure 4.13: Operational scheme of XWRT.

XWRT. XML Word Replacing Transform [SGS08] follows a similar idea to that proposed by XMill, since it also considers the separation between structure and data content, and data division into several containers, but grouping them regarding the element names, not the whole path from the document root. Notwithstanding, the most important difference that in fact constitutes the backbone of XWRT, is the use of a dictionary, obtained in a preliminary pass over the document, to replace the most frequent words with index references. Dictionary entries are encoded using a byte-oriented prefix code, optimized for further compressions (e.g. *gzip*, LZMA, etc.). Yet XWRT also applies specific encodings for different numerical data (e.g. sequence of digits, dates, bibliographic information, fractional numbers, etc.). In this case, the numerical value is replaced with a flag in the main output stream, while the actual value is encoded and sent to the corresponding container. Finally, all encoded results are passed to general compressor schemes, namely *gzip*, LZMA or PPM, yielding to the compressed XML file. In Figure 4.13, XWRT general operational scheme is depicted.

XComp. XComp [Li03] consumes XML data and produces the output in a streaming fashion. It constitutes another example of an XML compressor that applies the principle, first introduced by XMill, of structure and data separation, but with slight modifications, that are following presented.

The structure refers to the different *markups* of the XML document, while data refers to data associated to these markups. Authors consider as basic markups tags and attribute names, but they also take into account special markups as processing instructions, comments, CDATA sections, etc. whose data are string

Code	**Meaning**
0	End-tag
1	Data item position
2	'=' position (only used when preserve whitespace)
3	'>' position (only used when preserve whitespace)
4	Whitespace position (only used when preserve whitespace)
5	The position of any characters before XML Declaration
6	PI
7	DTD
8	Comment
9	CDATA section

Figure 4.14: Markups codification used by XComp.

values between their limiters. While parsing the XML document, structure is separated from data. To represent the structure, the different markups are encoded by using an integer codification. Regarding the corresponding data, this structure representation only records data items positions, that will be then grouped and separately stored in other containers. Each different tag and attribute name is assigned a different numerical identifier starting from 10. The integers ranging from 0 to 9, are reserved to indicate special markups and notations, whose meaning are shown in Figure 4.14. For instance, value 1 is used to indicate the positions of data items. Hence, the encoded structure of the XML document fragment of Figure 4.15 a), using the code assignment of Figure 4.15 b) will result $10, 11, 11, 12, 1, 0, 13, 11, 12, 1, 0, 0, 14, 11, 12, 1, 0, 0, 0$. In addition, XComp also stores data items lengths in a separated array. Therefore, and assuming the same example, the lengths kept for data items *D0824*, *2011*, *The Descendants*, *A0173*, *George Clooney*, *A0128*, and *Shailene Woodley* are $5, 4, 15, 5, 14, 5$, and 16, respectively. This structure representation follows a model where white spaces are not preserved. However, XComp also provides a model where they are considered.

Note that this structure representation is similar to that used by XMill, with the exception that in case of attributes, XComp saves specifying an identifier for the data item corresponding to an attribute name[4], since it realizes that in every well-formed XML document it will always be present, and hence attribute value identifiers are implied by those of attribute names.

On the other hand, and with respect to data content, XComp follows a semantic-like approach, where data is grouped not only based on their tag/attribute names, but also based on their level (depth) in the document tree and their type. That is, data items are sent to a same container if they share the same tag/attribute name, the same level, and the same node type (i.e. a tag or an attribute). For instance, in the example of Figure 4.15 a), `ID` will result into two containers, as well as `name`,

[4] Observe that there is no 1 directly after any integer identifying an attribute name.

a) **XML Document**

```
<movie ID="D0824" year="2011">
   <name>The descendants</name>
   <actor ID="A0173">
      <name>George Clooney</name>
   </actor>
   <actress ID="A0128">
      <name>Shailene Woodley</name>
   </actress>
</movie>
```

b) **Code assignment**

10	movie
11	name
12	actor
13	actress
14	ID

Figure 4.15: Tag/attributes identifiers (b) assigned by XComp to compress a sample XML document (a).

while the rest of the tags and attributes will lead each one to one different container. This is done based on the idea that, data with the same name, but at different levels or of different types, may have different domains or formats and hence also have different semantics and distributions. XComp also has special containers to store data items from processing instructions, comments, and so on. In any case, all data items are stored as strings in the containers.

In a final compression step XComp applies one of two optional compression schemes, namely, *gzip* or Huffman, to integers from structure and data length containers, and dictionary structure container, and also to the strings of each individual data item container. This step can be performed when the document parsing has finished, but also when a *memory window* size is exceeded, since to obtain an efficient memory usage, XComp sets a maximum space size for the containers (that can vary their sizes along the process). When this limit is reached, data of the different containers are sent to the compression engine, and the result is streamed to the output. In case of Huffman coding, statistical information is gathered when parsing the document for each individual container, and a Huffman tree is also written to the output by the compression engine.

XBzip. This compressor is an adaptation of the XBW transform [FLMM05, FLMM09], inspired by the Burrows-Wheeler transform (BWT) for strings [BW94], to represent succinct labeled trees. XBzip [FLMM06, FLMM09] constitutes the tool to obtain a simple compressed and non-searchable representation of an XML document, based on the XBW. Yet the same authors also created XBzipIndex [FLMM06, FLMM09], the compressed searching and navigable version, further detailed in Section 4.2.3.

One of the main characteristics of XBW transform is that its own construction leads to an automatic grouping of the contexts (i.e. paths), in contrast with other XML conscious compressors, that explicitly separate them in order to compress

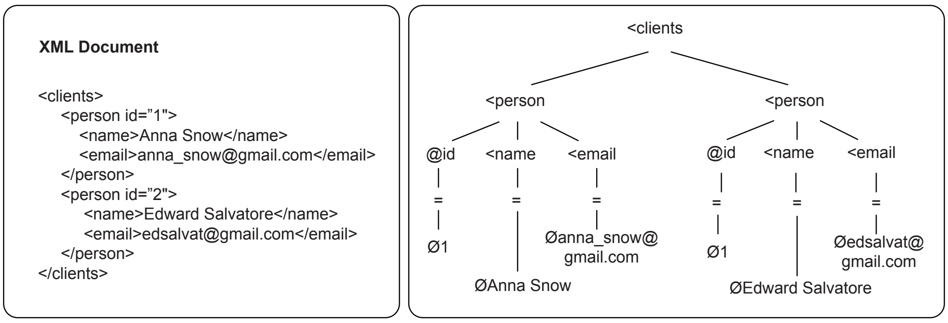


Figure 4.16: An XML document (a) and its corresponding ordered labeled tree (b).

together *similar* ones. To obtain the XBW transform of an XML document, this is first modeled as an ordered labeled tree T, where each occurrence of a start-tag, $\langle t \rangle$, or an attribute name, att, originates a node labeled by $\langle t$ and $@att$, respectively, and where both attribute values and text content, say δ, are replaced by two nodes, one labeled with $=$, and the other one, with $\emptyset\delta$, being $\emptyset$ a symbol not occurring elsewhere in the XML document. We assume for forthcoming explanations, that this T representation has t nodes, from which n are internal nodes, and l are leaves, thus $t = n + l$. Figure 4.16 shows the ordered labeled tree of a sample XML document.

In a second step, a *sorted multiset* S of triplets $\langle S_{last}, S_\alpha, S_\pi \rangle$ is built (one for each tree node), by traversing T in pre-order and by generating for each visited node, i, the corresponding triplet $S[i] = \langle S_{last}[i], S_\alpha[i], S_\pi[i] \rangle$. The first component of that triplet is a binary flag set to 1 if and only if i is the rightmost child of its parent, the second component is given by the label of i, and the third one, is the upwards labeled path from i parent to the root of T. Once all the triplets are obtained, they are sorted with respect to the third component, and finally the XBW transform is composed by three arrays[5] $\langle \widehat{S}_{last}, \widehat{S}_\alpha, \widehat{S}_{pcdata} \rangle$, where $\widehat{S}_{last} = S_{last}[1, n]$, $\widehat{S}_\alpha = S_\alpha[1, n]$, and $\widehat{S}_{pcdata} = S_\alpha[n{+}1, t]$. In Figure 4.17 we show the XBW transform construction from the T representation of Figure 4.16. Notice that as BWT groups together characters prefixed by the same substring, XBW does the same regarding the data enclosed in the same upwards path.

The final step of XBzip, consists of storing the arrays $\langle \widehat{S}_{last}, \widehat{S}_\alpha, \widehat{S}_{pcdata} \rangle$, in a compact way. For this purpose, $\widehat{S}_{last}$ and $\widehat{S}_\alpha$ are merged in an unique array $\widehat{S}_{\alpha'}$, and then both $\widehat{S}_{\alpha'}$ and $\widehat{S}_{pcdata}$, are separately compressed by using the PPMdi [Shk02] compressor scheme.

[5]This XBW transform differs from the original one, defined in [FLMM05] as the pair $\langle \widehat{S}_{last}, \widehat{S}_\alpha \rangle$, to better exploit XML documents features.

S_{last}	S_α	S_π
1	<clients	*empty string*
0	<person	<clients
0	@id	<person<clients
1	=	@id<person<clients
1	Ø1	=@id<person<clients
0	<name	<person<clients
1	=	<name<person<clients
1	ØAnna Snow	=<name<person<clients
1	<email	<person<clients
1	=	<email<person<clients
1	Øanna_snow@gmail.com	=<email<person<clients
1	<person	<clients
0	@id	<person<clients
1	=	@id<person<clients
1	Ø2	=@id<person<clients
0	<name	<person<clients
1	=	<name<person<clients
1	ØEdward Salvatore	=<name<person<clients
1	<email	<person<clients
1	=	<email<person<clients
1	Øedsalvat@gmail.com	=<email<person<clients

S_{last}	S_α	S_π
1	<clients	*empty string*
0	<person	<clients
1	<person	<clients
1	=	<email<person<clients
1	=	<email<person<clients
1	=	<name<person<clients
1	=	<name<person<clients
0	@id	<person<clients
0	<name	<person<clients
1	<email	<person<clients
0	@id	<person<clients
0	<name	<person<clients
1	<email	<person<clients
1	=	@id<person<clients
1	=	@id<person<clients
1	Øanna_snow@gmail.com	=<email<person<clients
1	Øedsalvat@gmail.com	=<email<person<clients
1	ØAnna Snow	=<name<person<clients
1	ØEdward Salvatore	=<name<person<clients
1	Ø1	=@id<person<clients
1	Ø2	=@id<person<clients

Stable sort

XBW Transform

$\hat{S}_{last}$ = 10111110010011111111

$\hat{S}_\alpha$ = <clients<person<person====@id<name<email@id<name<email==

$\hat{S}_{pcdata}$ = Øanna_snow@gmail.comØedsalvat@gmail.comØAnna SnowØEdward SalvatoreØ1Ø2

Figure 4.17: The set S after the pre-order traversal of T (left) and after its stable sort regarding the component S_π (right), together with the final output of the XBW transform (bottom).

Exalt. Based on the fact that an XML document can be defined by a context-free grammar, Exalt [Tom03] consists of a syntactical compression scheme that uses the grammar-based codes encoding technique [KY00] to incrementally generate the grammar, which is then encoded by an adaptive arithmetic coding [WNC87]. But prior to this, Exalt tries to exploit the redundancy of the XML document structure, and to derive predictions that may substantially improve compression efficiency. That is called the *structure modeling* of the document.

The goal of the *structure modeling* is to reduce the amount of data to be next appended to the underlying compression scheme. To this aim, numeric tokens are used to represent the structure (as done, for instance, by XMill or XComp), in such a way that both character data content and numeric tokens are passed to the grammar-based coder to be compressed together (unlike other solutions where data compression follows a container-based approach). Numeric tokens capture the redundant information, like repeated appearances of tags and attributes, but also of special events such as end-tags, the beginning of a comment, an entity declaration, a processing instruction, etc.

Moreover, while processing the document, Exalt also aims to learn as much as possible about its structure. Most times elements present quite a regular structure, hence the main idea is to retain it and to use this knowledge to predict their future *structural* behavior. In case the prediction is successful no symbols need to be generated, thus reducing the amount of data to be compressed. Therefore, each XML element will be assigned a finite state automaton, called *model of the element*, which describes its structure. The automaton states can be either *element* or *character* states, representing nested elements and contained character data, respectively. Transitions between states describe the composition of nested elements and character data within the element, and keep frequency counters that are then used to compute the most probably transition, based on their probability. Element models are adaptive. Initially, they consists of an initial state with no transitions, which are incrementally added together with new states and also updated, as the data is processed. In this way, element models are used to make predictions of the structure. Each time an element is processed, prediction succeeds if the expected state of its model (that is, the state ending the most probable transition) really happens. In that situation, we only need to update the counter of the predicted transition and then enter the referenced model, but without sending any data to be encoded. Yet the models may give wrong predictions, if elements have an irregular structure. In those situations, an escape event is produced in conjunction with the information needed to correct the prediction.

AXECHOP. AXECHOP [LDM05] is an XML-conscious compression scheme that combines a grammar-based compression of document structure with a Burrows-Wheeler Transform [BW94] compression of the data portions of a document. Hence structural and data parts are divided, following the idea introduced by XMill. The structure of the XML document is first transformed by applying a byte tokenization scheme, that preserves the original structure of the document, and then a context-free grammar is produced by using the MPM compression algorithm [KYNC00]. This grammar is finally compressed with an adaptive arithmetic coder [WNC87]. Regarding the data content, different containers are created according to the specific tag/attribute enclosing the data, and then the Burrows-Wheeler Transform is applied to each separated container.

4.2.3 Queriable XML Compressors

Queriable XML compressors are usually schema-oblivious tools, since they equally consider both to obtain reasonable compression ratios, and to provide some kind of query support. Therefore, a most interesting division considers their homomorphic or non-homomorphic nature, rather than schema awareness.

4.2.3.1 Homomorphic Compressors

As it has been previously seen, homomorphic compressors retain the original configuration of an XML document. They are not as common as non-homomorphic ones. Yet, compressors like XGrind and XPRESS have become some of the most representative tools within the queriable XML compressors category.

XGrind. It constitutes the first XML-conscious compressor able to support queries over the compressed form, that is, without the need of a full decompression of the compressed XML document. XGrind [TH02] makes it possible thanks to its *homomorphic* nature, that does not serrate structure from data content, leading to a compressed document that preserves the syntactic structure and semantics information of the original document. In fact, the compressed XML document can be viewed as the original one, but replacing tags, attributes and their respective values by the corresponding encodings. Hence available techniques or even indexes [MWA+98] for processing regular XML documents, can be similarly built on the XGrind compressed output.

To compress a given XML document, XGrind uses different encoding techniques depending on the kind of token:

- *Structural tokens*: whenever a start-tag is encountered, it is replaced by a 'T' followed by an uniquely identifier associated to the tag name. All end-tags are encoded by '/', while each occurrence of an attribute name applies a similarly encoding scheme than that used to code start-tags, but using the character 'A' instead of 'T'. The identifiers of the tag/attribute names are dictionary encoded, hence they represents indices to specific entries of a dictionary.

- *Enumerated-type attribute value tokens*: enumerated-type attribute values may be usual in XML documents. This kind of information is provided by the DTD of the XML document. Hence, if it is available, XGrind identifies which attribute instances hold this characteristic, and encodes their values by using a $log_2 K$ encoding scheme to represent the different K values that conform the enumerated domain.

- *Element/Attribute values*: since XGrind aims to an efficient query evaluation over the compressed document, it requires a *context-free* compression scheme to allow direct searches over the compressed document. Therefore, XGrind uses the classical Huffman coding [Huf52], but computing separated character-frequency distributions for each element and non-enumerated attribute, instead of using a single one for the entire document. Given that element/attribute values are usually semantically related, they are expected to have similar distributions.

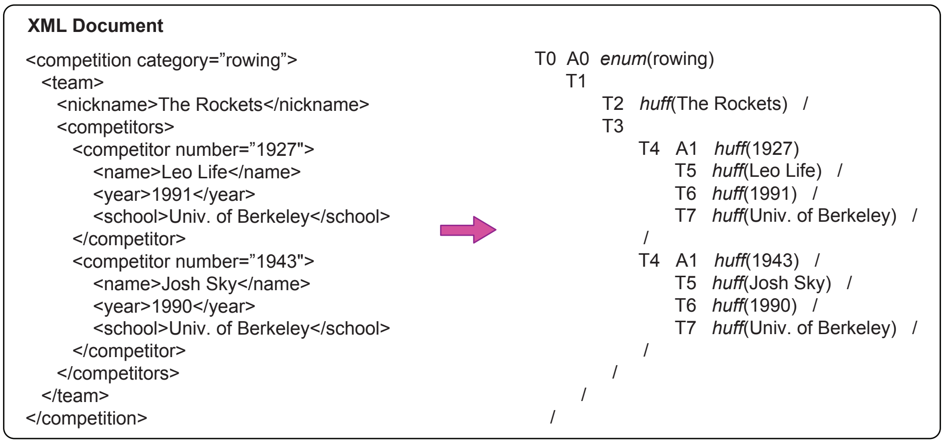


Figure 4.18: Abstract view of XGrind compression.

As a result, XGrind makes two passes over the input XML document to compress it. In the first one, different element and attribute names are gathered to be dictionary encoded, and also statistics for the element content and attribute values are collected to create the coding models of the different Huffman coders associated with elements and non-enumerated attributes. Regarding the enumerated-type attributes, and if the DTD is provided, corresponding code values are generated, as well, following the encoding scheme explained above. If not, they are coded applying the Huffman technique. Finally, in the second pass, document is compressed by encoding each token with the corresponding code, obtained in the first pass. Figure 4.18 shows an example of an XML fragment and its compressed version using XGrind. Note that *huff(s)* represents the output of the Huffman compressor for an input s, while *enum(v)* denotes the output of applying the corresponding encoding scheme for an enumerated attribute value v.

With the scheme applied by XGrind, *exact-match* and *prefix-match* queries can be performed over the compressed document without decompressing it. This is done by first compressing the query string and then searching for its corresponding encoded sequence in the compressed text. Nevertheless, partial decompression is still necessary for queries involving *range* or *partial* matches. Moreover, many other operations, like joins or nested queries, are not directly supported.

XPRESS. Like XGrind, XPRESS [MPC03] is another example of *homomorphic* compressor, hence preserving syntactic and semantic information of the original XML document, and supporting direct querying over the compressed version of the document. Again, different encoding schemes are used to compress the different token types appearing in an XML document. For instance, each element and attribute name is encoded by using a technique called *Reverse Arithmetic Encoding*. Inspired by arithmetic encoding [Abr63], this technique is designed for coding each of the aforementioned tokens regarding their whole tree path from the root of the document, by using real number intervals in the range $[0, 1)$. That is, each number interval represents the encoded element/attribute path. This feature leads to an important property: let us suppose two labeled paths, $P = p_i \dots p_n$, and $Q = p_j \dots p_n$, if $i >= j$, that is P is a *suffix* of Q, then this encoding scheme guarantees that the interval representing P, say I_P, contains the interval that represents Q, namely I_Q. With respect to content values, they are separately compressed using different *context-free* compression methods depending on their data type. For example, numerical values are compressed by applying *differential encoding* to their binary representation. In turn, enumerated-type values are encoded using a dictionary encoding, while the rest of the data values are compressed by using a Huffman encoder [Huf52]. As happened in XGrind, the XPRESS compression procedure consists of two passes over the text. The first one is devoted to compute statistics, and the last actually compresses the document. Figure 4.19 represents a conceptual view of the resulting compressed document after using XPRESS, over the XML fragment shown in Figure 4.18. Observe that rac_i denotes the output of coding an element/attribute tree path with *Reverse Arithmetic Coding*. Likewise, *enum(v)*, *huff(s)*, and *num(m)* stands for encoded elements and attribute values regarding the different coders used according to their data type.

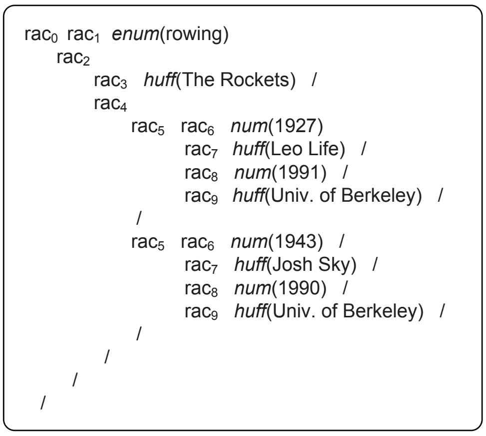


Figure 4.19: Abstract view of XPRESS compression.

Although it applies similar ideas to XGrind, XPRESS improves XGrind twofold. On the one hand, it encodes complete tree paths instead of just individual element/attribute names. Moreover it uses an encoding scheme that satisfies *suffix* containment. By this way, path-based queries are evaluated straightforward by simply checking the interval containment between the path of the posed query and those of the compressed document, without need of decompression. For instance, if the path of a query is `//team/name`, then the query processor will select elements `/competititon/team/competitors/name`, since the interval of `//team/name` will contain the interval of `/competition/team/competitors/name`. On the other hand, and given that numerical data values are encoded by using a compression technique that preserves order, *range* queries concerning numerical data can also be performed over the compressed document, unlike XGrind. However, *partial* matches and *range* queries not involving numerical values, still suffer from partial decompression.

QXT. QXT is an enhanced version of XWRT [SGS08], able to support query evaluation with partial decompression. Hence the main features present in XWRT are also kept in QXT. For instance, structure and data content are separated, the latter being additionally divided into several containers. Frequent words (including elements, attributes, and general data values) are replaced with index references to the entries of a dictionary, created in a first pass over the input document. Dictionary entries are encoded by using a byte-oriented prefix code. Regarding numerical data values and special data such as dates, times, and fractional numbers, they are coded with specific encoders and sent to the corresponding containers. All containers are then further compressed with general back-end compressors (e.g. deflate, LZMA, etc.). Nevertheless two main differences stand out from QXT. The first one is related to data containers division. Against XWRT, containers are created depending on the whole path from the document root, instead of only considering element names. The second feature is that containers are compressed in blocks of 32KB, thus allowing partial decompression of small data units. Query execution in QXT first tries to solve which containers might contain data matching the query. Then it decompresses the required containers, and finally the obtained transformed representation is searched also using the transformed pattern. As happened in XGrind and XPRESS, the set of different queries supported by QXT is still limited. QXT does not maintain any indices to document content since its primary purpose is effective compression.

4.2.3.2 Non-homomorphic Compressors

Together with schema independent non-queriable XML compressors, non-homomorphic queriable compressors are the categories from which more tools have been developed during the last years. Some representative methods of the second group are the 11 following tools:

XCQ. XCQ [LNWL03, NLWL06] is an XML schema-aware compressor based on a technique called *DTD Tree and Sax Event Stream Parsing* (DSP), that tries to takes advantage of the information provided by the XML document Document Type Definition (DTD) to generate concisely compressed data, but also useful to perform query evaluation. The DSP technique separates document structure and data content from the input SAX event stream produced while parsing the XML document. Similarly to those XML compressors that use the knowledge of a schema specification (like Millau, SCA, XAUST, etc.), it only encodes the structural information that can not be inferred from the DTD, that is, occurrences of $*$, $+$, ? and | operators. On the other hand, data part is arranged applying a *path-based* partition grouping. Each time data values are encountered, they are sent to the data stream associated with the full tree path connecting the data to the root node. In addition, these data streams are then divided into indexed blocks. Both, structure stream and blocks of data streams are finally individually compressed using a general text compressor, usually *gzip*.

Data block division slightly worsens compression ratio due to data commonalities that are limited to the contents of the current block. However, since blocks can be compressed and decompressed as individual units and given that they are created in a path-based manner, it also makes possible to only decompress those blocks that are relevant for a posed query. Therefore, a critical feature of XCQ is to determine the accurate block size, given that compression and query performance would be inversely affected.

XCQ supports the evaluation of a subset of XPath queries involving not only selection and predicates, but also aggregation operators (e.g. *count*, *sum*, *average*, etc.) and equality comparisons (e.g. $=$).

XQzip. XQzip [CN04] introduces indexing structures to support a wide range of XPath queries over the compressed XML document, although partial decompression is still needed for the matching of string conditions. XQzip separates structure (i.e. tags and attributes[6]) from data (i.e. element content and attribute values) while parsing the XML document. The first stream is used to build the *Structure Index Tree* (SIT), an indexing structure that removes duplicate structures from the XML document to improve query performance. In Figure 4.20 b) an example of a SIT is illustrated, which corresponds to the tree structure of the XML fragment of Figure 4.20 a). In turn, data are first grouped into different containers according to their associated tag/attribute, and then further divided into smaller data blocks which are separately compressed using *gzip*. These blocks can be decompressed individually, hence avoiding full decompression in query evaluation. Yet this leads to a trade-off between compression ratio and decompression overhead when querying, as happened in XCQ. If the block size is small, redundancies across separated blocks are not properly used, while if a large block size is defined it will be costly to decompress it. Hence, it may be difficult to find a suitable block size for both compression and query evaluation. To minimize decompression overhead in query evaluation,

[6]Namespaces, processing instructions and comments are not modeled by XQzip.

XQzip applies the Least Recently Used (LRU) algorithm to manage a buffer pool for the decompressed data blocks, thus avoiding repeated decompressions if the data is already in the pool. XQzip addresses different types of XPath queries, such as multiple predicates with mixed value-based and structure-based query conditions, but it also allows comparison (e.g. =, >, <, >=, <=, etc.), string (e.g. *contains* and *starts-with*) and aggregation operators.

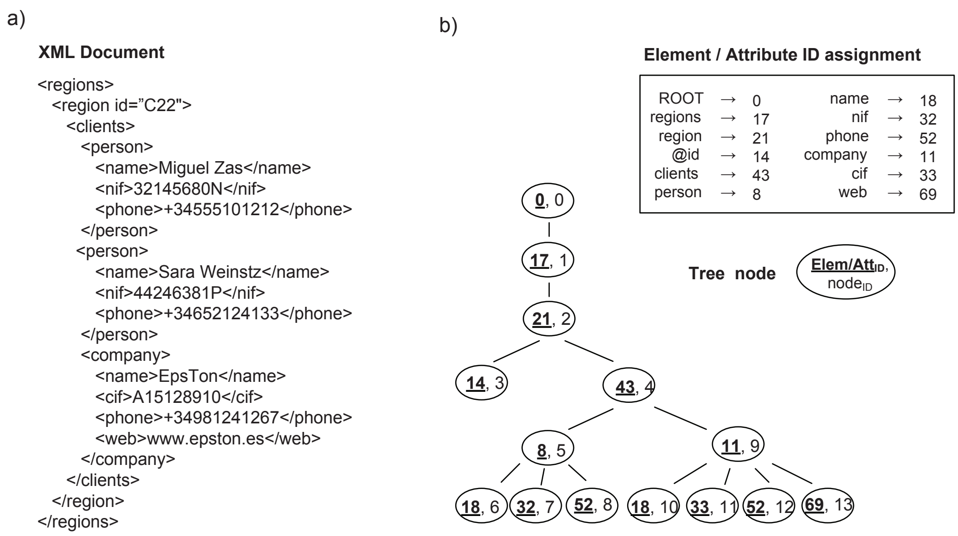


Figure 4.20: SIT structure (b) of an XML document fragment (a).

XMLZip. This compressor [XMLb] takes as input the DOM tree representation of an XML document, and it basically divides that tree into different components by pruning it at a certain depth, d, that can be specified by the user. Then each component is separately compressed with *gzip*. The component that contains all the nodes in the tree up to depth d is called the *root component*. The rest ones are *child components* and correspond to all the sibling subtrees starting at depth d. These children are replaced into the *root component* by references. Figure 4.21 shows an example of the DOM tree component division performed by XMLZip using $d = 2$. XMLZip does not improve compression ratios, compared with those obtained by compressing the document with the underlying *gzip*, yet its main advantage is that XMLZip supports partial decompression, by decompressing the portions of the compressed components that are needed for query evaluations.

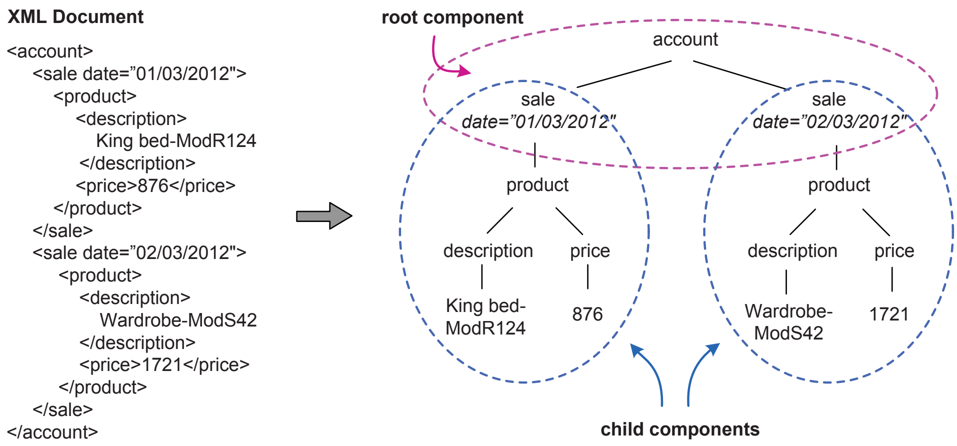


Figure 4.21: DOM tree division in XMLZip.

XQueC. This compressor [ABMP07] focuses on query speed rather than compression efficiency. As XGrind and XPRESS, XQueC compresses individual data items of the XML document to avoid decompression during query processing, but if differs from them on the separation of document structure and data parts. With respect to structure, tag and attribute names are encoded using a binary representation of $log_2 N$ bits, being N the total number of different names. Furthermore, XQueC builds a *structure tree* of the input XML document, where each node is assigned an unique identifier reflecting the order of the represented tag/attribute in the document and also the corresponding assigned code. Meanwhile, data values specified by the same root-to-leaf path are grouped into a same container. XQueC can choose to compress the XML data by applying either the ALM algorithm [Ant97], or the classical Huffman compressor [Huf52]. In the former situation, order is preserved in the encoded data, thus allowing one to perform range queries directly over the compressed values. In turn, Huffman algorithm supports prefix-wildcards (although not inequalities). Moreover, XQueC considers containers grouping into sets according to their contained data common properties to improve compression efficiency. To determine containers association, as well as the appropriate choice of the suitable compression algorithm, XQueC creates cost models of the different possible configurations by exploiting query workloads information.

XQueC supports a wide subset of XQuery language. To this aim, it also builds additional data structures and indices. For instance, it creates a dataguide [GW97], that is, a structural summary representing all possible paths in the document, and links each node to the corresponding data container. What is more, XQueC links each individually compressed data item to its corresponding node in the structure tree. Those auxiliary structures significantly improve query performance, however they may incur in a huge space overhead.

XSeq. XSeq [LZLY05] is another example of grammar-based compressor. It is based on Sequitur [NMW97a, NMW97b], a linear-time on-line algorithm that generates a context-free grammar that uniquely represents the input string. XSeq uses this algorithm to compress each of the several containers in which structure and data tokens of an input XML document have been previously separated. In addition, XSeq makes use of a set of indices to correlate data values stored in different containers, thus improving querying efficiency. For instance, a *header* index, pointing to each different container, and a *structural* index, through which each data value can be quickly located in the container without decompression. Data containers also include devoted indices. All those features grant to XSeq the ability of directly processing queries (in particular, XPath queries) over the compressed document, without full or partial decompression. XSeq is also able to process only relevant data values for a given query, thus avoiding a sequential scan of irrelevant compressed data.

XCPaqs. This compressor [WLLH04] separates structure and content, and compresses them separately. For the structural part, individual tags, but also complete root-to-leaf paths are considered. XCPaqs gathers statistics for both components, and it first codes tags with Huffman compressor [Huf52]. Then paths, which can be described as a series of tags in Huffman code, are further encoded, by using again the same encoder. Connection between structure and content is kept by the path order in the original document associated to each data. When processing the document, path type (i.e. data type and range of values of the data associated to a same root-to-leaf path) is recognized, in such a way, that data is compressed by using a specific compressor depending on the corresponding inferred path type. For instance, enumerated-type data are dictionary encoded, while string data are encoded with a suffix compressor, and long text is compressed with the Burrows-Wheeler Transform [BW94]. The obtained results from structure and content encoders are finally combined based on their connection relations, leading to a 2-ary final structure.

XCPaqs can solve XQuery queries. Before query processing, tags in the query are translated into their corresponding code and then the query plan is split into three steps: *i*) to select appropriate path codes; *ii*) to relate elements and conditions according to their content; *iii*) to construct the final result.

ISX. ISX [WLS07] proposes a compact storage scheme for XML, providing at the same time, efficient support for XPath query evaluation, and also update operations like insertions and deletions. ISX distinguishes three different storage layers: the *topology* layer, the *internal node* layer, and the *leaf node* layer. The first one stores the tree structure of the XML document by using a balanced parentheses encoding derived from [KM90]. The internal node layer, in turn, stores the elements, attributes and signatures of the data content for enabling fast text queries. Finally, data values are actually stored in the leaf node layer. Those data are referenced by the topology layer and can be compressed by various common compression techniques (usually *gzip*). Additionally, ISX creates auxiliary data structures over the basic storage scheme to allow efficient query processing.

TREECHOP. All procedures in TREECHOP [LMD05] visualize the input XML document as a tree structure, where non-leaf nodes correspond to elements and attributes, but also to CDATA sections, comments and processing instructions. In turn, leaf nodes are character data, such as attribute values and data content enclosed by an element. TREECHOP compresses the XML document in an adaptive way. As tokens are received by a SAX parser, new tree nodes are created and sent to the compression stream. Each non-leaf node is assigned a binary codeword. This codeword is uniquely assigned based on the complete path from the root of the tree node. Hence, nodes with the same absolute path, will receive the same codeword. Formally, the codeword C_n assigned to a non-leaf node n, with parent node p, is formed by the concatenation of three codes C_p, G_n, and T_n. C_p, represents the codeword of p, while G_n is a Golomb code [Gol66] assigned to n based on its order with respect to p. Finally, T_n, is a sequence of 3 bits denoting the kind of node (e.g. an element, an attribute, a comment, etc.). This encoding scheme keeps the structure of the original XML document. Regarding the leaf nodes, they are processed in a similar manner, using in addition reserved byte values to indicate the beginning and end of the associated character data. As node information is added to the compression stream, it is compressed using *gzip*. Like XGrind, TREECHOP supports *exact-match* queries through a sequential scan over the compressed document, while *range-match* queries require data values decompression to be further validated.

LZCS. Although it yields into this category, LZCS [ANF07] can not be considered a general purpose XML compressor, since it is specifically adapted to compress highly structured XML documents, and hence it does not perform well with arbitrary ones. Inspired by the Ziv-Lempel compression, LZCS replaces identical subtrees by a pointer to their first occurrence. To improve compression the *LZCS transformation* of a document can be further compressed with a classical compressor. In particular, authors use the semi-static word-based Huffman method [Mof89] and two PPM schemes [CW84], namely PPMdi and PPMz. The former keeps LZCS transformation properties related to navigation ability, while the latter does not. In [ANF09], authors show how to perform some basic XPath operations (regarding *child*, *descendant*, *parent*, and *ancestor* axes, and also text matching operator) over the LZCS transformation, by using a streaming approach. The main idea is to speed up path matching operations by taking advantage of the work done over repeated substructures.

XBzipIndex. As first disclosed in Section 4.2.2.2, XBzipIndex is the compressed and searchable tool of the XBW transform adaption presented in [FLMM06, FLMM09]. Like XBzip, the XBW transform computation of an XML document, given by $\langle \widehat{S}_{last}, \widehat{S}_{\alpha}, \widehat{S}_{pcdata} \rangle$, constitutes the first step of XBzipIndex construction. But to keep navigation and searching purposes, it also needs to support *rank* and *select* operations over $\widehat{S}_{last}$ and $\widehat{S}_{\alpha}$. Hence these two arrays are stored by using a compressed representation supporting the aforementioned operations (see [FLMM09] for more implementation details). In turn, $\widehat{S}_{pcdata}$, is first split into *homogeneous* buckets, in such a way that two elements are held in the same bucket if they have the same upward path, and afterwards a *FM-index* [FM01, FM05] representation is created for each bucket. Under this representation, XBzipIndex allows answering two different kind of queries: *i)* $//\Pi$, *ii)* $//\Pi[fn : contains(., \gamma)]$, where Π denotes a fully-specified path consisting of tag/attribute names and γ is an arbitrary string.

One of the distinctive features of XBzipIndex is that it constitutes the first solution combining compression and indexing. The compressed data represents at the same time the structured text and an index built on it. That is called a *self-index* [NM07].

SXSI. Like XBzipIndex, *Succinct XML Self Index* (SXSI) [ACM+10] is another tool for compressed indexing of XML data. Yet it is able to support a wider range of XPath queries than that addressed by XBzipIndex. SXSI is tailored to work in main memory, and uses a compressed index representation for XML data able to solve queries involving some of the *forward* XPath axes, together with different text functions (e.g. `'='`, `contains`, and `starts-with`).

SXSI regards XML documents as both an ordered set of strings, and also as a labeled tree defined by the hierarchical tags. Hence, it establishes a separation between the structure itself and the text content. Figure 4.22 illustrates the model used by this proposal for a given XML fragment. Note that the actual tree is formed by the solid edges, whereas dotted edges show the connection with the textual parts. Each node of the tree representing an element is labeled by its corresponding tag name, text nodes are modeled as leaves labeled with `#`, and each attribute node is represented as a sequence of nodes where the first one is labeled with `@`, its child node is the attribute name itself and the leaf child denotes the associated attribute value by means of the special label `%`. Observe that there is exactly one text content related to each tree leaf labeled `#` or `%`. Nodes of the tree are assigned *global identifiers*, but also each text content receives its own *text*

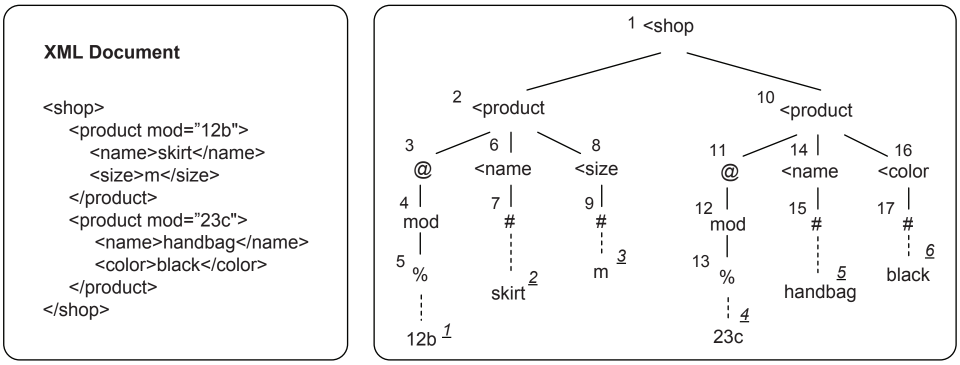


Figure 4.22: Example of SXSI data model.

identifier. Then, SXSI concatenates all text data[7] and represents them by using a succinct full-text self-index, namely the *FM-index* [FM05]. This index is based on the BWT [BW94] and supports pattern matching operations[8]. In turn, the tree structure is represented by combining two different and aligned sequences: a *balanced parentheses* representation of the tree skeleton, and a sequence of the tag identifiers of each tree node. Tree navigation operations are directly inherited from the implementation of the first sequence [SN10]. Figure 4.23 shows how SXSI models the structural and textual parts of the example depicted in Figure 4.22.

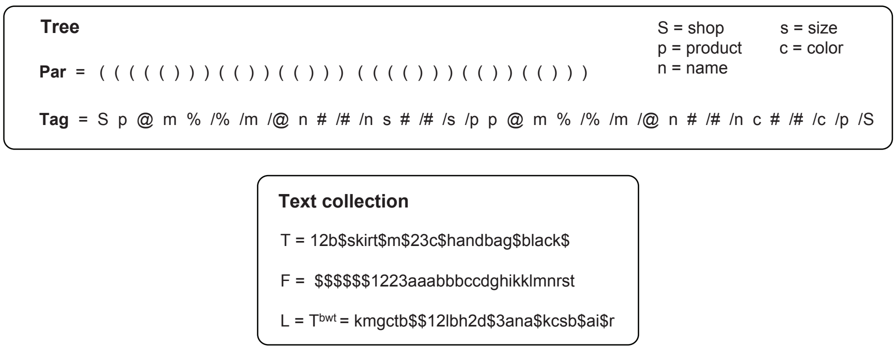


Figure 4.23: Tree and text data representation in SXSI.

The aforementioned data structures constitute the base for query evaluation. Each XPath query is translated into an alternating tree automaton [CDG+07,

[7]Each one appended with the special end-marker **$**.

[8]In addition, SXSI also stores the texts in plain format, to enable faster text extraction.

Hos10]. Conventionally, the run of a tree automaton visits every node of the input tree, but SXSI makes use of the information kept on the indexes and applies different techniques to only visit the relevant ones [MN10], thus reducing processing times.

Part II

Our proposal: XXS

Chapter 5

The XML Wavelet Tree

In this chapter we present the first core part of XXS, the XML Wavelet Tree (XWT), a new data structure to represent an XML document in a compressed and self-indexed way (see Figure 5.1). The XWT constitutes a new approach for compact representation of XML documents, which takes about 30%-40% of the original document size, allowing at the same time their efficient processing and querying: XWT provides implicit indexing properties that can be successfully profited to efficiently support XPath queries, as it will be later seen from Chapter 7 to Chapter 9.

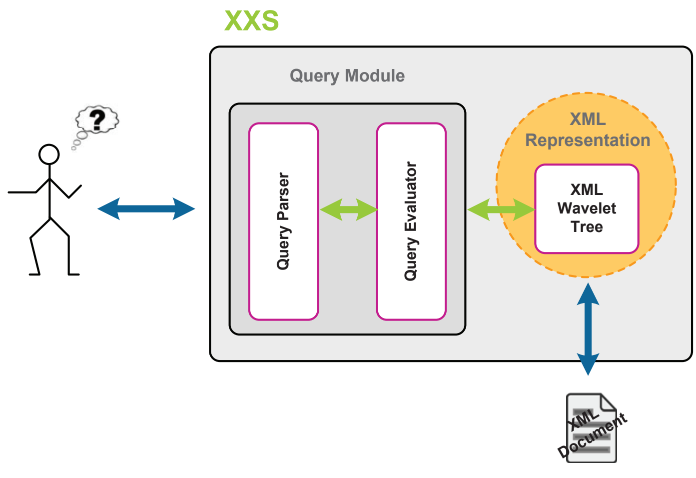


Figure 5.1: XML representation of XXS: the XML Wavelet Tree (XWT).

This chapter focuses on the XML Wavelet Tree data structure description. Section 5.1 first introduces the main construction features of this representation, while Section 5.2 details the basic procedures to decompress and search over the XWT. Sections 5.3 and 5.4 end the chapter by uncovering some of the main XWT properties that lead to an efficient query support.

5.1 XWT Construction

Following the essence of the WTBC reorganization of codewords strategy explained in Section 3.1.8, XML Wavelet Tree has been specifically designed to deal with XML documents and to efficiently support XML retrieval, by especially focusing on XPath queries.

Although WTBC can be applied to any word-based, byte-oriented semistatic statistical compression technique, XWT uses the *(s,c)*-Dense Code compressor described in Section 3.1.4.3 (the reason of that choice will be explained next). As a result, the process of obtaining the final XWT representation of an XML document is made in two phases. Making a first pass on the source text, the first phase obtains its different words[1] and frequencies (the *model*), and assigns codewords to each word according to an *(s,c)*-Dense Code encoding scheme. Then, in a new pass on the source text, the second phase replaces each word of the text by its codeword, leading to a compressed representation of the text. But these ones are not stored consecutively. Codewords are placed along different nodes following a WTBC organization.

Inside this general construction process, many different and important features are considered, to make XWT suitable for efficient querying purposes.

5.1.1 Phase I: Parsing the XML Document and Assigning Codewords

5.1.1.1 Document Parsing

The first step is to parse the input XML document to gather the different words that will compose the vocabularies and to compute their frequency distribution. To this aim, we use a variant of the spaceless word model [MNZBY98].

The *parsing* process distinguishes different kind of words depending on whether a word is[2]:

- A *start-tag* or an *end-tag*.

[1] We speak of words to simplify the discussion. In practice both words and separators are encoded as atomic entities in word-based compression.

[2] Division implicitly given in accordance with the XPath data model [XPaa].

- The name of an attribute.
- An attribute value.
- A word inside a *comment*.
- A word inside a *processing instruction*.
- A word of the XML document text content.

In some cases, this distinction arises from the same XML document construction features, with special markups that signal each kind of word. In the other cases, the differences will be internally maintained when parsing. Note that, to hold this, the basic spaceless word model used is slightly modified. In the basic spaceless word model, tokens are based on alphanumeric and non-alphanumeric character types, in such a way that contiguous strings of similar characters are isolated. In our parsing, we keep this, but not in a strict sense, since we also consider the following cases as single words independently of the fact that alphanumeric and non-alphanumeric characters are mixed:

- The group of characters formed by the left angle bracket, <, and the name of a start-tag markup: <name
- The end-tag markup as a whole: </name>
- The name of an attribute followed by the equal character: name=
- The reserved initial and final characters groups defining a special markup, such as *comments* (<!-- and -->), *processing instructions* (<? and ?>), *CDATA sections* (<![CDATA[and]]>), etc.

As a result, when compressing, a same word will be assigned different codewords depending on the category it belongs to. For example, if the word book appears as text content (e.g. ...the great book ...), but also as an attribute value (e.g. category=“book") and inside a comment (e.g. <!-- ...this book is ...-->) it will be stored as three different entries in the vocabularies, one for each different category, leading to three different codewords.

Keeping this difference between words sharing a common name according to their role in the XML document increases the vocabulary size, however it will be shown that this translates into efficiency and flexibility when querying.

It is also when parsing that some *normalization* operations take place (all according to [XMLa]). For instance, empty-element tags are translated into their corresponding pair of start-end tag (e.g. <author/> becomes <author> </author>). While keeping satisfied the well-formedness constraints according to [XMLa], this uniformity in the representation maintains both the boundaries of the tags and the structure relations of the document perfectly defined. We also consider some other minor considerations with no relevant meaning to document processing like the removal of redundant spaces and spaces inside tags (e.g. `<author     >` becomes `<author>`).

Taking the previous classification into account, four different vocabularies are created while parsing the XML document:

- The *content* vocabulary, which holds words from the text content category together with attribute value entries[3].
- The *tags* vocabulary, keeping the different start-tags and end-tags.
- The *attributes* vocabulary, which stores word entries corresponding to attribute names.
- The *nsearch* vocabulary, holding words appearing inside processing instructions and comments.

Notice that the first two vocabularies are always present. The rest of the vocabularies will be created or not, depending on the presence or absence of attributes, processing instructions, and comments into the particular XML document being parsed. Henceforth, we also refer as *special* vocabularies those apart from the *content* vocabulary. Figure 5.2, shows an example of a XWT representation[4] built from an XML document sample, for which the four different vocabularies are created.

5.1.1.2 Codewords Assignment

To assign codewords, we use *(s,c)*-Dense Code as the compression technique. Remember that it uses different bytes for *continuers* and for *stoppers*. Therefore, by reserving one of the *continuers* to be the first byte of the codewords assigned to words of the *special* vocabularies (one different *continuer* for each of the vocabularies), we can gain important benefits. These benefits arise from the fact that, by enforcing this encoding, words from each of the previous vocabularies are all kept located under same branches of the XWT, that is, they are isolated.

For instance, if we consider the example of Figure 5.2, where byte b_3 is the *continuer* reserved to be used as the first byte for all the codewords assigned

[3] Notice that although attribute values and text content words share a same alphabet, different word entries are stored in case of same words appearing in both categories, hence receiving different codewords. For example, in Figure 5.2, the word `love` appears as an attribute value, but also inside the text content of `opinion` tag. Therefore, we keep two different entries inside the *content* vocabulary (see $\texttt{love}_{\texttt{att}}$ and $\texttt{love}_{\texttt{text}}$ entries).

[4] Note that only the shaded byte sequences are stored in the XWT nodes; the text is shown in the figure only for clarity.

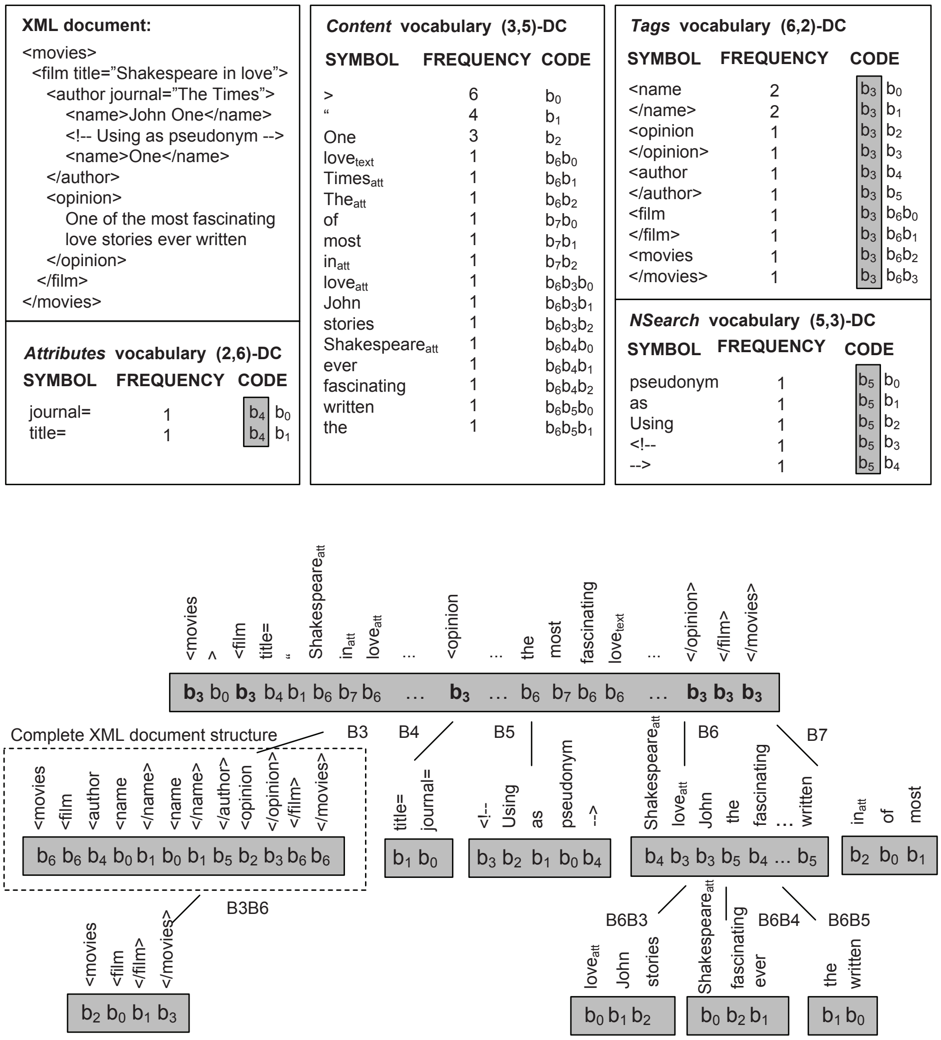


Figure 5.2: Example of XWT structure built from an XML document.

to words from the *tags* vocabulary (see the bytes shaded in the `CODE` column of the *tags* vocabulary), we will notice that the branch $B3$ (and also its children) is devoted to exclusively store start-tags and end-tags. Remember that they follow the document order, and hence maintain their relationships like in the original XML document, so actually we can say that branch $B3$ stores the complete structure of the XML document. Moreover, and as it will be further detailed in Section

5.3, this branch exactly matches a balanced parentheses representation of the XML document structure. Hereafter, we will refer as *XDTree*[5] the XWT node holding the complete structure of the XML document.

The same idea of using a specific starting byte can be extended to the rest of the *special* vocabularies. In Figure 5.2, attribute names are stored under branch $B4$, and words from processing instructions and comments, under $B5$. In the former case the isolation gives the flexibility needed in XPath to directly operate over attributes, while in the second one it allows one to easily distinguish fragments that should be skipped in general text searches. Notice that the remaining words kept in the rest of the branches of the XWT structure (i.e., text content and attribute values, entries of the *content* vocabulary) are those mainly involved in text matching procedures.

Therefore, once parsing has finished, the words of the *content* vocabulary are first assigned a codeword following an *(s,c)*-Dense Code encoding scheme, but keeping aside as many *continuers* as needed depending on the number of *special* vocabularies we have. Following with the example of Figure 5.2, where a $(3,5)$-DC encoding scheme is used to encode *content* words, the first three *continuers*, namely bytes b_3, b_4, and b_5, are disregarded. Notice that they are not used as a first byte of any of the codewords assigned to words of the *content* vocabulary.

Example To better understand this, let us assume that, as shown in Figure 5.2, we work with bytes of 3 bits (hence, $2^3 = 8$ different bytes are available, instead of 256 as usual), and that we use a $(3,5)$-DC to code *content* words, so *stoppers* are values between 0 and 2 (that is, between 0 and $s-1$) and *continuers* are values between 3 and 7 (that is, between s and $s+c-1$). If we reserve the first three *continuers* (i.e. 3, 4, and 5) to mark words of the *special* vocabularies, the codewords that could be assigned to *content* words are as follows: $\langle 0\rangle$, $\langle 1\rangle$, $\langle 2\rangle$ (one byte codewords), $\langle 6,0\rangle$, $\langle 6,1\rangle$,$\langle 6,2\rangle$, $\langle 7,0\rangle$, $\langle 7,1\rangle$, $\langle 7,2\rangle$ (two bytes codewords), $\langle 6,3,0\rangle \ldots \langle 6,7,2\rangle$, $\ldots$, $\langle 7,3,0\rangle \ldots \langle 7,7,2\rangle$ (three bytes codewords), and so on. Notice that those codewords starting with 3, 4, and 5 are skipped.

Because of this arrangement, compression could be affected[6], so to minimize compression loss, words of the *special* vocabularies are also coded following another *(s,c)*-Dense Code encoding scheme, according to their respective models. That is, optimal s and c values are computed for each of the *special* vocabularies. As a result, the codeword of a word of any of these vocabularies will always start by the reserved *continuer* of the corresponding vocabulary, to preserve isolation; but the remaining bytes of the codeword will follow the *(s,c)*-Dense Code scheme of the vocabulary that the word belongs to. Assuming again the example of Figure

[5] That stands for *XML Document Tree*.

[6] Notice that no word from the general vocabulary, that is, from the *content* vocabulary, can be encoded with a codeword starting by any of the reserved *continuers*. Hence, there will be useless groups of codewords.

5.2, we can see that codewords assigned to words of the *tags* vocabulary follow a $(6,2)$-DC encoding scheme, but keeping added as their first byte the reserved *continuer* b_3. In the same way, words of the *attributes* and *nsearch* vocabularies are coded by following a $(2,6)$-DC encoding scheme and a $(5,3)$-DC encoding scheme, respectively, but keeping the corresponding *continuer* (b_4 for attributes and b_5 for processing instructions and comments) as their first byte.

Example Following with the example detailed above, in which we assumed, as in Figure 5.2, bytes of 3 bits, the codewords that could be assigned to words of the *tags* vocabulary by using a $(6,2)$-DC encoding scheme and keeping the byte 3 as the first byte (the first *continuer* reserved from the $(3,5)$-DC encoding scheme used to code *content* words) are as follows: $\langle 3,0\rangle \ldots \langle 3,5\rangle$ (two bytes codewords), $\langle 3,6,0\rangle \ldots \langle 3,6,5\rangle, \ldots, \langle 3,7,0\rangle \ldots \langle 3,7,5\rangle$ (three bytes codewords), $\langle 3,6,6,0\rangle \ldots \langle 3,6,7,5\rangle, \ldots, \langle 3,7,6,0\rangle \ldots \langle 3,7,7,5\rangle$ (four bytes codewords), and so on. In case of codewords of the *attributes* vocabulary, that follow a $(2,6)$-DC encoding scheme and that have assigned the *continuer* 4 as their first byte, we distinguish: $\langle 4,0\rangle \ldots \langle 4,1\rangle$ (two bytes codewords), $\langle 4,2,0\rangle \ldots \langle 4,2,1\rangle, \ldots, \langle 4,7,0\rangle \ldots \langle 4,7,1\rangle$ (three bytes codewords), $\langle 4,2,2,0\rangle \ldots \langle 4,2,7,1\rangle, \ldots, \langle 4,7,2,0\rangle \ldots \langle 4,7,7,1\rangle$ (four bytes codewords), and so on. Finally, if we use a $(5,3)$-DC encoding scheme to code words of the *nsearch* vocabulary, in addition to the *continuer* 5, that must start any codeword of this vocabulary, the codewords that could be assigned to these words are as follows: $\langle 5,0\rangle \ldots \langle 5,4\rangle$ (two bytes codewords), $\langle 5,5,0\rangle \ldots \langle 5,5,4\rangle, \ldots, \langle 5,7,0\rangle \ldots \langle 5,7,4\rangle$ (three bytes codewords), $\langle 5,5,5,0\rangle \ldots \langle 5,5,7,4\rangle, \ldots, \langle 5,7,5,0\rangle \ldots \langle 5,7,7,4\rangle$ (four bytes codewords), and so on. Notice that the codewords of any word of the *special* vocabularies, are always composed by at least two bytes.

5.1.2 Phase II: Compressing and Creating the XWT Structure

Once codewords are assigned to words, we perform a second pass over the text replacing each word by its codeword and storing these codeword bytes along the different nodes of the XWT. The node where a byte of a codeword is stored depends on the previous bytes of that codeword, as explained in Section 3.1.8. Hence, the root of the XWT is formed by a vector with all the first bytes of the codewords, following the same order as the words they encode in the original document. Each node *BX* in the second level contains all the second bytes of the codewords whose first byte is b_x, following again the same order of the text. That is, the second byte corresponding to the j^{th} occurrence of byte b_x in the root, is placed at position j in node *BX*, and so on. For instance, in Figure 5.2, the eighth byte in the root is b_6, since $\texttt{love}_{\texttt{att}}$ is the eighth word of the text, and its codeword is $b_6b_3b_0$. The second byte of its codeword, b_3, appears in the second position of node *B*6 because

Algorithm 5.1: Construction of XWT

```
Input: d, an XML document
Output: XWT representation of d
// 1st pass
parseDoc(d, vocS, bp)
foreach voc ∈ vocS do
    sort(voc)
    computeOptimalSC(voc)
    codewordsAssignment(voc)
nodeS ← computeTotalNodes(vocS)
sizeNodeS ← computeSizeNodes(vocS, nodeS)
foreach node ∈ nodeS do
    XWT[node] ← allocate(sizeNodeS[node])
    track[node] ← 1
// 2nd pass
foreach w ∈ d do
    cw ← getCode(w)
    cNode ← root
    foreach i = 1 ... |cw| do
        pos ← track[cNode]
        XWT[cNode][pos] ← cw^i
        track[cNode] ← pos + 1
        cNode ← getChildNode(cNode, cw^i)
return concatenation of XWT nodes, size of each node sequence, vocabularies
together with optimal s values, bitmap to construct the balanced parentheses
representation of the XML document structure
```

`love`$_{\texttt{att}}$ is the second word in the text encoded with a codeword starting by b_6. In turn, its third byte is the 1^{st} one of node $B6B3$ because its second byte is the first b_3 in node $B6$.

The XWT nodes can be allocated and filled with the codeword bytes as the second pass takes place, because it is possible to precompute the number of nodes as well as their sizes in advance (more precisely, just after the first phase is finished). So, by only keeping an array of markers indicating the next writing position for each node, they can be sequentially filled following the order of the words in the text.

At last, the compressed text is generated as the concatenation of the sequences of each XWT node, plus a header with their sizes. The XWT data structure generated also includes the different vocabularies, and their respective optimal s values (taken from the corresponding *(s,c)*-Dense Code encoding schemes), together with the bit array representation of the XML document structure (created while parsing the document, by setting a 1-bit for each start-tag and a 0-bit, for each end-tag), from which its balanced parentheses representation can be later built.

Algorithm 5.1 shows the pseudocode of the global construction procedure of a XWT representation. It takes an XML document as input, and yields as output the XWT data structure generated.

5.2 XWT Basic Procedures

As it has been shown in Section 5.1, the XML Wavelet Tree constitutes a compressed representation of an XML document. Still, it also provides some *implicit* indexing properties that make this structure be self-indexed as well. It occupies a space proportional to the compressed document, but it implicitly allows one to perform some searching operations more efficiently than over the typical plain compressed version.

The two basic procedures using the XWT are to recover any word at a specific position of the document, and to search for a pattern. They both are performed with simple traversals over the XWT tree by using *rank* and *select* operations, respectively. Original codewords can be rebuilt from the bytes spread along the different XWT nodes by using *rank* operations, while words can be efficiently located, taking advantage of the self-indexing properties of XWT, by using *select* operations. Thereby, the efficiency of the XWT hinges on the implementation of the *rank* and *select* operations. In this thesis, we use the particular implementation described in Section 3.2.1.2 that uses a structure of partial counters to speed up *rank* and *select* operations. Next, we will explain how the two basic procedures and some other primary ones are performed over the XWT, by dividing them into two different blocks depending on whether they provide *decompression* or *searching* capabilities.

5.2.1 Decompression

5.2.1.1 Random Word Decompression

If we want to decode a word at any position of the document we will use *rank* operations and perform a top-down traversal of the XWT. Hence, to extract a random document word j (*random decompression*), we first access the j^{th} byte of the root node of the XWT to get the first byte of its codeword. If according to the encoding scheme it is the last byte of a codeword (that is, it is a *stopper*), we finish the procedure. However, if the codeword has more than one byte (since the read byte is a *continuer*), we will continue traversing the XWT top-down to get the rest of the bytes. Notice that, at this point, we have to check if the byte read, b_i, matches any of the *continuers* reserved to mark the codewords of any word of the *special* vocabularies. Depending on this, going down in the XWT to obtain the remaining bytes will be done by using the s and c values of the corresponding vocabulary. Whatever the case, by reading b_i as the first byte, we already know that the second byte of the codeword is stored in the node Bi. As all the words whose codeword starts by byte b_i will have their second bytes placed at node Bi, we only have to count how many times the byte b_i occurs in the root node until position j. So we compute $rank_{b_i}(Root, j) = k$, that tells us that the second byte of the codeword we are decoding is the k^{th} byte of node Bi. Again, if that byte is not yet the last one (that is, if it is not a stopper), we proceed in a same way until the last byte of the codeword is reached. Algorithm 5.2 depicts the pseudocode to decode a word at a given position of the document.

Algorithm 5.2: *Display* text position x

Input: p, a position of the document
Output: w, word at position p in the document

$cNode \leftarrow root$; $cw \leftarrow \emptyset$
$b \leftarrow XWT[cNode][p]$
$cw \leftarrow cw \; || \; b$
while b *is a continuer* **do**
  $p \leftarrow rank_b(XWT[cNode], p)$
  $cNode \leftarrow getChildNode(cNode, b)$
  $b \leftarrow XWT[cNode][p]$
  $cw \leftarrow cw \; || \; b$
$w \leftarrow getWord(cw)$
return w

Example To know which is the third word in the source document of Figure 5.2, we will proceed as follows. We start by reading the third byte of the XWT root, that is, we get $Root[3] = b_3$. According to the encoding scheme, we know that byte b_3 is a *continuer*, but also that it is one of the reserved *continuers*, in particular, that reserved to mark *tags* codewords. So, on the one hand, we know that the codeword is not complete yet, and we will have to read a second byte in the second level of the XWT, more precisely, in node $B3$, since it holds all the codewords starting by b_3. On the other hand, we also know that hereafter the process will continue by using the encoding scheme associated to the *tags* vocabulary. Therefore, the next step will be to find out which position of node $B3$ we have to read. By using $rank_{b_3}(Root, 3) = 2$ we obtain that there are 2 bytes b_3 in the root until position 3. Thereby, $B3[2] = b_6$, gives us the second byte of the codeword we are looking for. Again b_6 is not a *stopper*, so we need to continue the procedure. In the child node $B3B6$, that corresponds to the first two read bytes of the codeword we are decoding, we have to read the byte at position $rank_{b_6}(B3, 2) = 2$. Finally, we obtain $B3B6[2] = b_0$. But b_0 is a *stopper* and, therefore, it marks the end of the searched codeword. The complete codeword is $b_3b_6b_0$, corresponding to the start-tag `<film`, which is precisely the third word in the document, as expected.

5.2.1.2 Full Text Extraction

If we want to decompress the whole document from the beginning (*full decompression*), we can proceed by extracting each word individually. However, we can take advantage of a more efficient procedure. Full decompression implies sequentially covering the bytes of the root node and getting the codewords whose first byte is stored there. Then the same process as the previous seen to decode a word could be applied from the beginning of the root, $j = 1$. But, given that the sequences of bytes of all the XWT nodes follow the original order of the words in the source document, *full decompression* can be efficiently implemented using pointers to the next positions to be read in each node. That is, when going to a child node to read the following byte of an uncomplete codeword, we do not need to compute any *rank* operation to find out which position of this child node sequence we have to read. It always will be the next one to process in that child node. The pseudocode for *full decompression* is described in Algorithm 5.3.

Algorithm 5.3: *Full text extraction*

```
Output: d, original XML document
foreach node ∈ nodeS do
    track[node] ← 1
d ←∅
foreach i = 1 ... sizeNodeS[root] do
    cNode ← root; cw ←∅
    b ← XWT[cNode][i]
    cw ← cw || b
    while b is a continuer do
        cNode ← getChildNode(cNode, b)
        b ← XWT[cNode][track[cNode]]
        cw ← cw || b
        track[cNode] ← track[cNode] + 1
    d ← d || getWord(cw)
return d
```

We will illustrate this procedure with the example of Figure 5.2. The first step consists of initializing an array, we call *track*, that holds the positions of the first unprocessed entry of each XWT node with the value 1. Then we start by reading the byte at position 1 in the root node. Since it is a *continuer*, b_3, we know that the codeword is not complete, so we have to move to the second level of the tree, in particular, to node $B3$, and read the second byte of the codeword. It is at this point that, by using the basic decoding procedure of a word, a *rank* operation is performed to know which position of node $B3$ should be read. Instead, we just have to read the byte of node $B3$ at the position given by $track[B3] = 1$. Therefore, we obtain the byte b_6, and we update the value *track*$[B3]$ to 2. Again, according to the encoding scheme, the codeword is still not complete, so we proceed in the same way, but in node $B3B6$. That is, we read the byte of that node placed at position *track*$[B3B6] = 1$, which is byte b_2, and then update the value of *track*$[B3B6]$ to the next unprocessed entry of that node, 2. Since byte b_2 is a *stopper*, we can get the first decoded word, corresponding to the codeword $b_3b_6b_2$, the word `<movies`. After that, we continue with the second word at the root node. The byte of the root node at position 2 is b_0. It is the last byte of a complete codeword, so we finish the decompression of the second word of the document, by obtaining the corresponding word, `>`. The following word at the root node is the third one. At position 3 in the root node, we get the byte b_3, hence we have to read a second byte of the codeword in node $B3$. Since the value of *track*$[B3] = 2$, we know that this second byte of the codeword we are searching for is at position 2 in node $B3$. So we read byte b_6, that newly leads us to node $B3B6$. Thus, we first update the value of *track*$[B3]$ to 3, and then we proceed in an analogous way, but in node $B3B6$. Finally, we obtain the third word of the document, `<film`. We will continue the same procedure, until reaching the last word of the root node, saving unnecessary *rank* operations and making faster the complete document decompression.

5.2.1.3 Partial Decompression

The same smart procedure applied to efficiently perform *full decompression* can be extended to extract a fragment of the document starting from a random position, instead of from the first one. The only difference lies in the initialization of the *track* array, since we do not start by decompressing the document from the first position. A priori, we cannot know the first unprocessed entries of each XWT node, so when decompressing a word, whenever the *track* value of a visited node is uninitialized, we will perform a *rank* operation to set the value of the next byte to be read. Otherwise, we will just read the byte at the position given by the corresponding *track* entry. That is, at most we have to pay one *rank* operation for each XWT node, because once a XWT node has been visited, *rank* operations are avoided.

Notice also that this mechanism can be used, for instance, to speed up snippet extraction around a found occurrence of a word, by just applying the procedure from the earlier position of the snippet in the root node of the XWT up to the last one.

5.2.2 Searching

5.2.2.1 Word Patterns

Locating. In general, we can find the position in the document of any occurrence of a word by first searching for the last byte of its codeword in the corresponding XWT node, and then performing consecutive *select* operations up to the root of the XWT. This procedure arises from the own organization of codeword bytes. Given a codeword $\langle cw^1...cw^m \rangle$, if byte cw^i occurs at position j in the corresponding XWT node (that is, in node $B_{cw^1}B_{cw^2}...B_{cw^{i-1}}$), then the previous byte of that codeword, cw^{i-1}, will be the j^{th} one occurring in the parent node (that is, in node $B_{cw^1}B_{cw^2}...B_{cw^{i-2}}$). Therefore, when the root node is reached, we have the position of the word into the document. This procedure is sketched in Algorithm 5.4.

Algorithm 5.4: *Locate* j^{th} occurrence of word w operation

```
Input: w, a word; j, an integer
Output: pos, position of the j^th occurrence of w
cw ← getCode(w)
cNode ← computeLastNode(cw) // the node where cw^|cw| is placed
pos ← j
foreach i = |cw| ... 1 do
    pos ← select_{cw^i}(XWT[cNode], pos)
    cNode ← getParentNode(cNode)
return pos
```

For instance, let us assume we want to locate the first occurrence of $\texttt{love}_{\texttt{att}}$ in the example of Figure 5.2. The codeword of this word is $b_6b_3b_0$, then we have to start the search at node $B6B3$, since b_6b_3 are the first bytes of the codeword till the last one. Next, we will search in which position of node $B6B3$ the first byte b_0 occurs (the last byte of $\texttt{love}_{\texttt{att}}$ codeword), by computing $select_{b_0}(B6B3, 1) = 1$. In this way, we obtain that it is at position 1, that is, the first occurrence of word $\texttt{love}_{\texttt{att}}$ is the first one of the words held in node $B6B3$ (i.e. words with codewords starting by b_6b_3). Also, we know that all the codewords whose last byte is stored in node $B6B3$, are represented in node $B6$ with a byte b_3, and that they are in the same text order. Therefore, the value 1 we obtained with the previous *select* operation indicates that the first byte b_3 in node $B6$ corresponds to the first occurrence of word $\texttt{love}_{\texttt{att}}$ in the document. Again, we compute $select_{b_3}(B6, 1) = 2$, that newly indicates that our codeword is the second one starting by b_6 in the root node. Finally, by computing $select_{b_6}(Root, 2) = 8$, we can answer that the first occurrence of $\texttt{love}_{\texttt{att}}$ is the 8^{th} word in the document.

To locate all the occurrences of a word, this procedure is repeated for each one. Since the traversed XWT nodes are the same for each occurrence and these will be processed consecutively, *select* operations and thus the whole process, can be sped up by using pointers to the already found positions in the XWT nodes.

Counting. To *count* the number of occurrences of a given word, is equivalent to compute how many times the last byte of the codeword assigned to that word appears in its corresponding XWT node. This node will be identified by all the previous bytes of the codeword. Therefore, in a general case, if a word is encoded with a codeword $b_x b_y b_z$ (being b_x and b_y, *continuers* and b_z, a *stopper*), it is only necessary to count the number of bytes b_z in node $BXBY$. That is, we only have to perform $rank_{b_z}(BXBY, i)$, where i is the size of the node $BXBY$. In turn, if the codeword has just one byte, b_z, we will do $rank_{b_z}(Root,n)$, where n is the number of words in the document, that is, the number of bytes in the root of the XWT. Taking the example of Figure 5.2, if we want to count the number of occurrences of `Shakespeare`$_{\texttt{att}}$, we have to first obtain its codeword, $b_6 b_4 b_0$, and then count the number of times its last byte, b_0, appears in the node identified by the first bytes of its codeword ($b_6 b_4$), that is, in node $B6B4$. In a same way, to count how many times the word `<name` appears in the document, given its codeword $b_3 b_0$, we only have to count the number of times the byte b_0 (since it is the last byte of its codeword) occurs in node $B3$ (since b_3 is the first byte of its codeword). Regarding words whose codeword has only one byte, like `One` in the same example of Figure 5.2, which is encoded by b_2, we only have to figure out how many times the byte b_2 (as it is the solely one, hence also the last byte of the codeword) appears in the root of the XWT (since all the first codeword bytes are placed in that node).

Algorithm 5.5: *Count* operation for a word w

Input: w, a word
Output: occ, number of occurrences of w

$cw \leftarrow getCode(w)$
$cNode \leftarrow root$
foreach $i = 1 \ldots (|cw| - 1)$ **do**
  $cNode \leftarrow getChildNode(cNode, cw^i)$
$occ \leftarrow rank_{cw^{|cw|}}(XWT[cNode], sizeNodeS[cNode])$
return occ

Algorithm 5.6: *Count* operation for a word w until a position p

Input: w, a word; p, a position of the document
Output: occ, number of occurrences of w up to position p

$cw \leftarrow getCode(w)$
$cNode \leftarrow root$; $occ \leftarrow p$
foreach $i = 1 \ldots (|cw| - 1)$ **do**
  $occ \leftarrow rank_{cw^i}(XWT[cNode], occ)$
  $cNode \leftarrow getChildNode(cNode, cw^i)$
$occ \leftarrow rank_{cw^{|cw|}}(XWT[cNode], occ)$
return occ

Notice that by applying this procedure, *count* operation turns into the search of a byte inside a node of the XWT, instead of searching for the occurrences of a word inside the whole document, hence the benefits are straightforward. Algorithm 5.5 shows the pseudocode of this operation. Moreover, we can also count the number of occurrences of a word until a given position of the document. In that case, we just perform the same strategy, but for each codeword byte, tracking down the endpoint toward the leaf node of the word. The pseudocode for that scenario is presented in Algorithm 5.6.

5.2.2.2 Phrase Patterns

Locating and counting. Apart from individual words, we may also be interested in locating several words, that is, in searching *phrase* patterns. To efficiently perform this over the XWT structure, we start by locating the first occurrence of the least frequent word of the pattern in the root node. Then we check if all the first bytes of the codewords of each word of the phrase pattern match the previous and next bytes of the root node. If those matches happen, we continue by validating the rest of the bytes of the corresponding codewords, until either we detect a false matching or we find the complete phrase pattern. But if it is not the case, we save going down in the XWT, and we simply locate the next occurrence of the least frequent word to be processed in a same way. This same basic procedure is used for both locating and counting a phrase pattern, and it is shown in Algorithm 5.7.

5.3 XWT Connection with a Balanced Parentheses Representation

As we briefly disclosed in Section 5.1.1.2, the *XDTree* node of XWT (node $B3$, in the example of Figure 5.2) provides a structural isolation, that establishes a biunivocal relationship between this node and a balanced parentheses representation (BP) of the XML document structure. This correspondence allows to combine both representations for an efficient evaluation of XPath queries.

The balanced parentheses representation supports in constant time a very complete set of tree operations (like finding the *parent*, the *open/close* pair, or even the *depth* of a node) given the position of a tree node (that is, a start-tag or an end-tag, in our case). Notice that a position in the BP matches the same position in the *XDTree* node. For instance, if we consider the BP representation of the example of Figure 5.2, i.e. (((()())())) (see Figure 5.3), we can observe that the third "(" is closed by the ")" placed at position eight, and that they precisely match the third and eighth byte entries of node $B3$, that are `<author` start-tag and `</author>` end-tag, respectively. Therefore, we can easily perform basic tree operations over the BP, and use the XWT to locate a position of the BP into the

Algorithm 5.7: *Count* operation for a phrase pattern ph

Input: ph, a phrase
Output: occ, number of occurrences of ph

$cph \leftarrow getCodes(ph)$
$order_{min} \leftarrow getLessFrequentWord(ph)$ // least frequent word position in ph
$total_{min} \leftarrow computeTotalOccurrences(cph[order_{min}])$ // number of occurrences of the least frequent word
$i \leftarrow 1$
while $i < total_{min}$ **do**
  $pos_{min} \leftarrow locateLessFrequentWord(cph[order_{min}], i)$
  $fail \leftarrow 0$
  foreach $j = [1 \ldots order_{min}) \cup (order_{min} \ldots |ph|]$ **do**
    if $cph[j]^1 \neq XWT[root][pos_{min} - order_{min} + j]$ **then**
      $fail \leftarrow 1$; **break;**
  if $!fail$ **then**
    foreach $j = [1 \ldots order_{min}) \cup (order_{min} \ldots |ph|]$ **do**
      $pos_j \leftarrow pos_{min} - order_{min} + j$
      $cNode \leftarrow root$
      foreach $k = 1 \ldots |cph[j]| - 1$ **do**
        $pos_j \leftarrow rank_{cph[j]^k}(cNode, pos_j)$
        $cNode \leftarrow getChildNode(cNode, cph[j]^k)$
        if $cph[j]^{k+1} \neq XWT[cNode][pos_j]$ **then**
          $fail \leftarrow 1$; **break;**
      if $fail$ **then break;**
    if $!fail$ **then** $occ \leftarrow occ + 1$
  $i \leftarrow i + 1$
return occ

original document (by simply going one level up from the matching position in the *XDTree* node through a *select* operation), or even to obtain its corresponding tag identifier (by applying the *decode* procedure described in Section 5.2.1.1, from the same position in the *XDTree* node), without the need of any additional data structure to hold that information.

Let us consider again the example of Figure 5.2 for better understanding of these powerful relationship. Suppose that we are searching for the first occurrence of the start-tag `<opinion`. By using the *locate* procedure of a word pattern explained in Section 5.2.2, we can obtain its location in the text (in the example, we can see that it is placed at position 33[7]), but also the location of its codeword bytes, as we perform consecutive *select* operations up to the root of the XWT. In case of the

[7] Reader can infer this position from the XML document fragment of Figure 5.2.

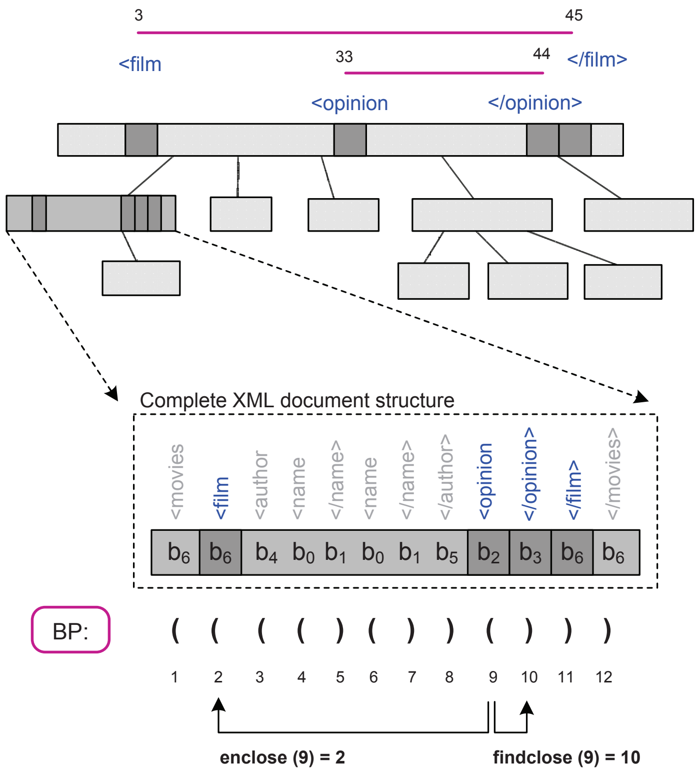


Figure 5.3: Example of correspondence between the *XDTree* node and a balanced parentheses representation (BP) of the XML document structure.

XDTree node, we can notice that `<opinion` corresponds to position 9, and that this position exactly matches the same location of `<opinion` in the BP representation, as stated. Now, let us assume two different scenarios[8] that could arise once that occurrence of `<opinion` is located (see Figure 5.3):

- *Locating the matching end-tag*: we might be interested in obtaining where the corresponding `<opinion` end-tag (that is, `</opinion>`) is placed. In that situation, we can take advantage of the *findclose* operation provided by the balanced parentheses representation, and just perform $findclose\ (9) = 10$, given that `<opinion` is at position 9 there. This tells us that the end-tag we are searching for corresponds to position 10 of both the BP and also the *XDTree*

[8]Those implying the use of *findclose* and *enclose* tree operations, since they are two of the most common ones.

node. Therefore, next, by simply computing $select_{b_3}(Root, 10) = 44$ (since the codeword of `</opinion>` is b_3b_2) we can also answer that `</opinion>` is the 44^{th} word of the text[9]. Last, we know that the first occurrence of *opinion* starts at position 33 and finishes at position 44 in the original document.

- *Discovering the parent tag*: another quite interesting situation may stem, for instance, from the need of discovering the identifier of the parent of `<opinion`. Notice that the typical *enclose* operation supported by a balanced parentheses representation provides that information, but from a positional standpoint. That is, given a position in the BP, this operation returns the position of the start-tag that encloses it. Hence, by computing *enclose* $(9) = 2$, we can first obtain that the parent of the target occurrence of `<opinion` corresponds to position 2 in the BP, but also in the *XDTree* node of the XWT. Therefore, once given this location, we simply need to perform the *decode* procedure described in Section 5.2.1.1 from that node, to obtain the complete codeword[10], and to finally decode the word corresponding to the start-tag parent of `<opinion`, which is `<film`.

5.4 Segments in an XML Document

Another important feature worth mentioning at this point is that given an element (tag), the positions of its corresponding start-tag and end-tag mark the limits of a *segment* in the XML document, which covers the text area enclosed by the element. For instance, see the segments depicted in pink on top of the XWT structure in Figure 5.3. However, this characteristic does not only apply for elements. In the same way, phrase patterns can be ultimately regarded as segments whose initial and final positions are given by the positions of the first and last word of the pattern, respectively. Indeed, even when working with words, we can also consider them as particular cases of segments, this time starting and finishing at the same position.

That is, any component of an XML document (e.g. an element, an attribute, a word, a phrase, etc.) could be ultimately regarded as a *segment*, $[s, e]$, whose limits arise from the *start* (s) and *end* (e) positions in the text, of the own component. Notice as well that, given two segments, a, $[a.s, a.e]$, and b, $[b.s, b.e]$, such a kind of representation allows one to compare them by using the relations shown in Figure 5.4.

[9] Reader can infer again this position from the XML document fragment of Figure 5.2.

[10] Notice that, in this case, the *decode* procedure could start from the second byte of the codeword, since we already know which is the first one. Remember that the *XDTree* node is devoted to just store the occurrences of start/end-tags, therefore all of them share the same first byte, that is, the reserved continuer b_3.

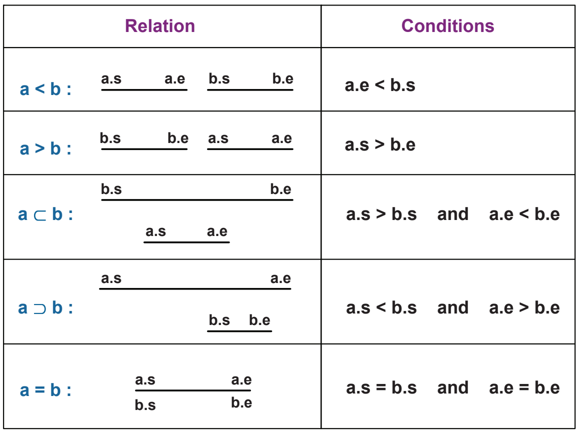


Figure 5.4: Segments relationships.

As it will be next discussed in Chapter 7, this *segment* representation will become a key factor to perform query evaluation over the XWT.

Chapter 6

Query Plan Construction

In Chapter 5 we presented the storage core of XXS, given by the XWT data structure. As we could see there, it constitutes a novel approach to represent XML documents in a compressed and self-indexed way. But more important is the fact that thanks to the self-indexing properties that this representation provides and its own construction features, query evaluation can be efficiently supported.

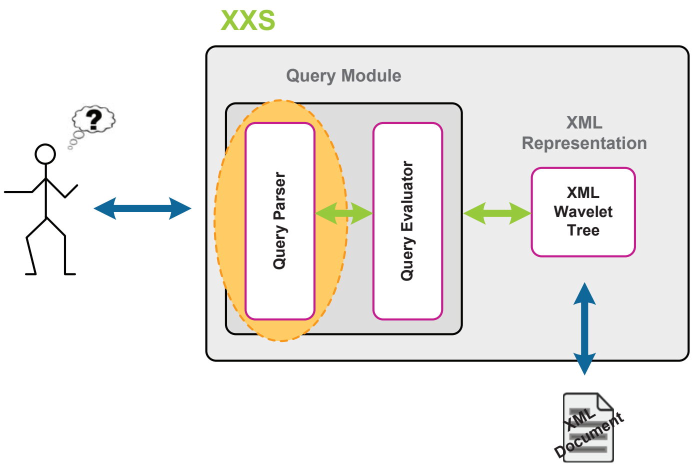


Figure 6.1: *Query parser* submodule of the XXS system.

The *Query module* of the XXS system evaluates XPath queries over XWT. This module is composed by two main components: the *Query parser*, and the *Query* *evaluator*, that are in charge of the query parsing and the query execution process, respectively. This chapter focuses on the *Query parser* submodule (see Figure 6.1). In this way, Section 6.1 first starts by introducing the set of XPath queries addressed in this work. Then, Section 6.2 shows how the input query is transformed into an initial representation, the *query parse tree*. Several transformations are applied over the *query parse tree* up to get an optimized plan, the *query execution tree*. All these transformations are presented in Section 6.3. Finally, Section 6.4 exhibits the general procedure performed by this submodule through a complete example.

6.1 XPath Query Support

XXS system supports a wide fragment of XPath, in particular a practical subset of the "Core XPath" defined in [GKP05]. We show below the EBNF notation of the target fragment, where `axis` stands for any *forward* or *reverse* axis, and `nodeTest`, is either a tag/attribute name or the wildcard '*'[1].

```
Core            ::=  LocationPath | '/' LocationPath
LocationPath    ::=  LocationStep('/' LocationStep)*
LocationStep    ::=  Axis'::'NodeTest |
                     Axis'::'NodeTest'['Pred']'
Pred            ::=  Pred 'and' Pred |
                     Pred 'or' Pred | LocationPath |
                     '(' Pred ')'
```

In addition, we implement two of the most common text functions of XPath 1.0, namely the *equality* (=) and *contains* (`contains()`) functions, plus the *count* node set function (`count()`).

6.2 Initial Query Plan: the Query Parse Tree

As stated in Chapter 2, XPath *path expressions* (also known as *location paths*) are regarded as sequences of *location steps*, where the result of the current step makes up the context for the next one. Previous and current location steps are related by the *axes*. Hence, it is possible to get an initial representation of an input query, we call *query parse tree*, produced from the own query parsing[2], by converting sequences of location steps into a composition of binary operators, whose operands are the corresponding *node tests* and the composition of the location path itself. That is,

[1] Node type tests can also be supported, but they are not addressed in this work.

[2] To parse the input query and to split it into its different components we have used the source code provided by Benjamin Piwowarski, based on his *soul* library (`http://sourceforge.net/projects/soulparsing/`).

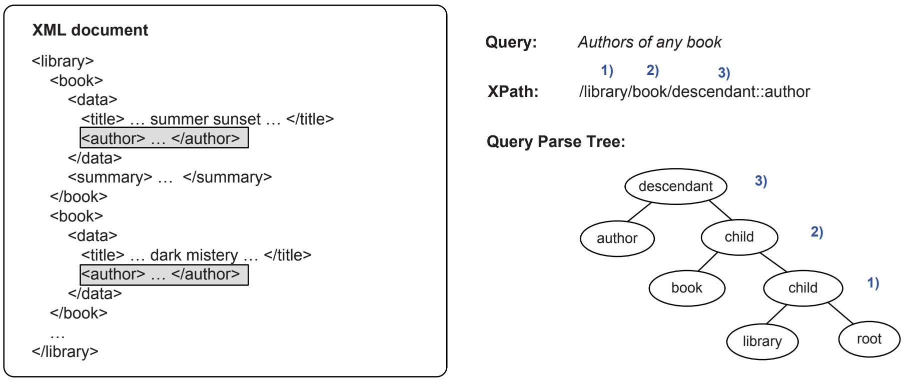


Figure 6.2: Example of query parse tree from a query without predicates.

by regarding the query from left to right, the query parse tree is built upwards as follows. Each location step is translated into a main node labeled with the step axis name and two children. The left child represents the location step node test, whose occurrences will be delivered by the main node (that is, its parent node in the *query tree*). In turn, the right child, is provided by the tree representation already set up from the previous location step. For instance, let us consider the query `/library/book/descendant::author`. Its query parse tree is depicted in Figure 6.2[3].

Regarding predicates, the location paths inside them can be similarly translated into a composition of binary nodes as the above mentioned paths outside predicates. This time, however, and to allow their further integration within the global query parse tree, we must reverse both the order in which the location steps are considered to build the tree (now from right to left) and the meaning of the axes. Axes with opposite meaning are, for instance, `child` ↔ `parent`, and `descendant` ↔ `ancestor`. Figure 6.3 illustrates two examples of query parse trees built from two different queries with predicates: *a*) `/library/book[./data/following-sibling::summary]/descendant::title`; *b*) `/library/book[contains(./descendant::title, “mistery”)]`. Observe that, in both cases, we can assume a separated parse tree for the predicate over `book` (see the parse trees inside the striped areas in Figure 6.3), obtained by a right-to-left traversal[4] of the predicate location path together with an axes reversal (namely, `following-sibling` ↔ `preceding-sibling` and `child` ↔ `parent` for query *a*); and `descendant` ↔ `ancestor`, for query *b*)), which is added to the general parse

[3] We refer as `root`, the *root node* of an XML document, according to the XPath data model.

[4] Note that in case of query *b*), the `contains` predicate would constitute the last step in a typical left-to-right traversal, since it is applied over `title`. Hence, if we consider the opposite traversal, it becomes the first step.

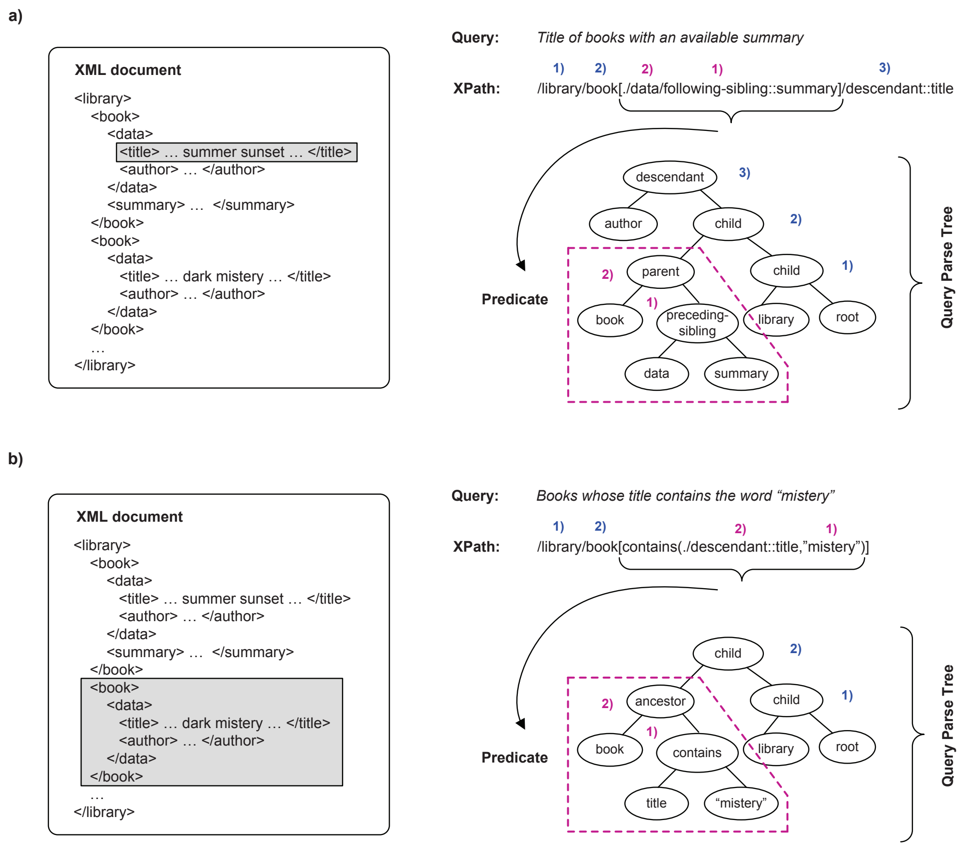


Figure 6.3: Example of query parse trees from queries with predicates.

tree when the second location step is processed to make up its corresponding left child.

6.3 Query Plan Optimization: Query Parse Tree Transformations

The initial parse tree of a query can be used as the *query execution tree* to be further evaluated. Nevertheless, we still perform some modifications over it to gain efficiency during evaluation. Some of them are simple algebraic simplifications, while some others are transformations that modify the original query parse tree (which only considers components of the XPath syntax), by producing an equivalent one in

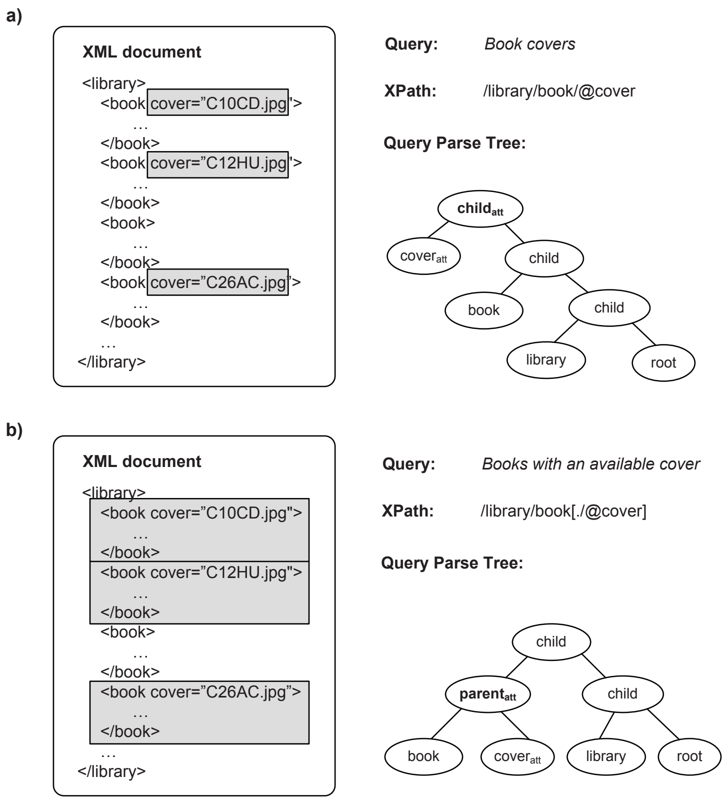


Figure 6.4: Examples of use of $\texttt{child}_{\texttt{att}}$ and $\texttt{parent}_{\texttt{att}}$.

terms of retrieved results, but optimized to meet XWT features. In fact, some of them are not intended to be general optimizations, rather they aim to get a better performance by creating a query execution tree tailored to exploit the characteristics of the XWT representation.

But prior to this, let us regard a notation that we will assume hereafter for better comprehension. We will use $_{\texttt{att}}$ to mark nodes representing attributes, but also to note nodes which stand for operators (i.e. axes/functions) any of whose child nodes is ultimately an attribute. This is done to make clear the difference between operators that may share the same name, but which at last result into different evaluation algorithms, depending on whether they are applied over an element or over an attribute, as it will be shown in Chapter 8. This notation applies for $\texttt{contains}_{\texttt{att}}$ and $\texttt{equal}_{\texttt{att}}$ text functions, but also for $\texttt{child}_{\texttt{att}}$ and $\texttt{parent}_{\texttt{att}}$. In particular, we use the two last ones to designate, respectively, the attribute selection of an element (see Figure 6.4 a)), and to select elements having a given attribute (see Figure 6.4 b)). Moreover, we will see that transformations may also lead to the $\texttt{descendant}_{\texttt{att}}$ and $\texttt{ancestor}_{\texttt{att}}$ operators, which are a generalization of $\texttt{child}_{\texttt{att}}$ and $\texttt{parent}_{\texttt{att}}$, respectively. That is, $\texttt{descendant}_{\texttt{att}}$ will select the attributes of an element or of any of its descendants, while $\texttt{ancestor}_{\texttt{att}}$ stands for elements that either have the target attribute or hold any descendant that has it.

Now, we will describe and exemplify the query parse tree transformations and also the different scenarios where they are applied, by dividing them into 4 main groups:

1. *Attributes equality simplification*: this modification consists of converting an equality step between an attribute name and its value, such as `...[@name="New York"]/...` or `.../@*[.="Spain"]/...` into a phrase matching operator, as shown in Figure 6.5.

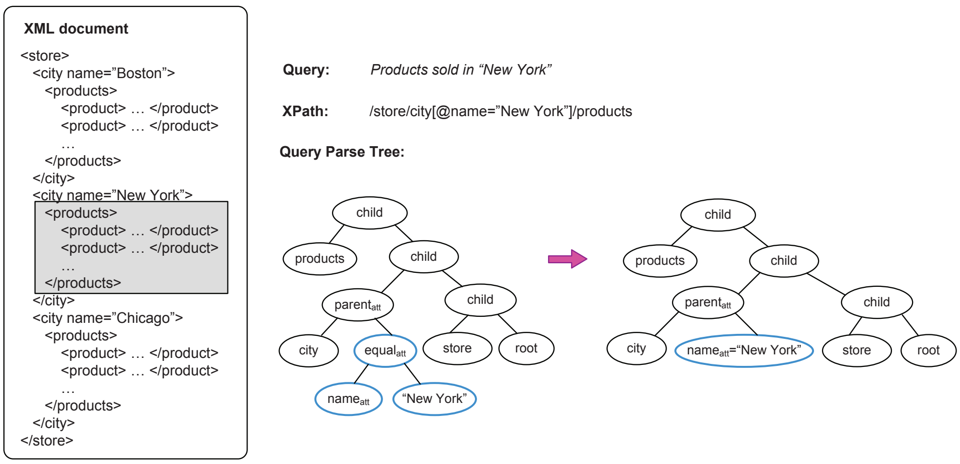


Figure 6.5: Example of *Attributes equality simplification.*

2. *Wildcard optimizations*: we distinguish the next three modifications over location steps involving wildcards (i.e., the asterisk wildcard `*`):

   (a) *Redundancy suppression*: this optimization aims at discarding a costly (or unnecessary) step. For instance, given the fragment of the query parse tree illustrated in Figure 6.6, we can avoid processing the `parent` step over the wildcard (which potentially selects all elements parent from an `editorial` element, to be further analyzed with respect to a

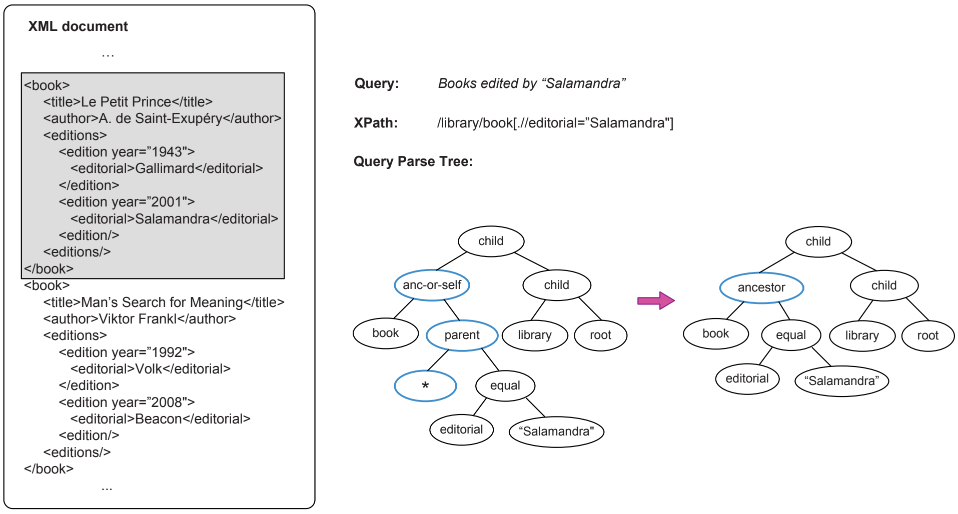


Figure 6.6: Example of *Redundancy suppression.*

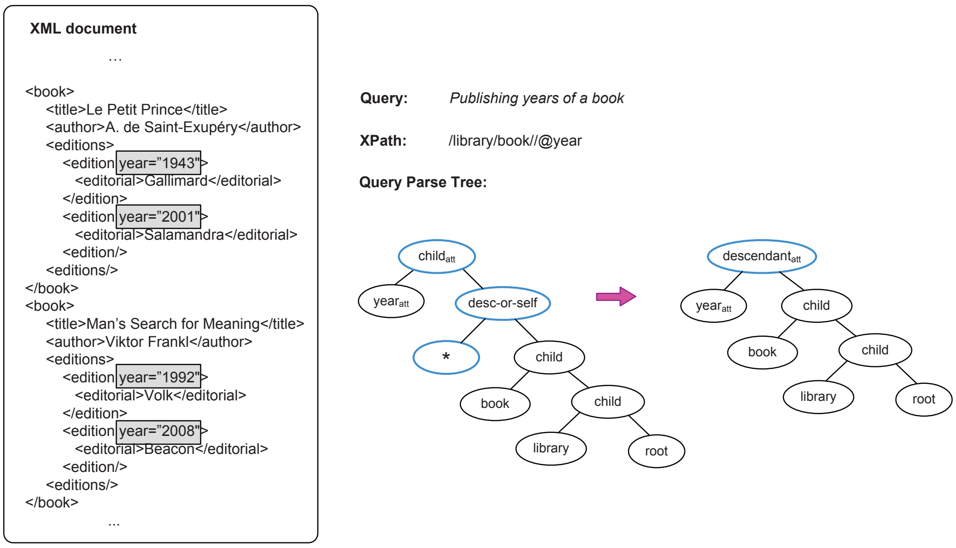


Figure 6.7: Another example of *Redundancy suppression.*

`book`), by combining it with the `ancestor-or-self` axis into a single step, `ancestor`. A similar scenario is shown in Figure 6.7. This time the involved axes are `descendant-or-self` and $\texttt{child}_{\texttt{att}}$. However,

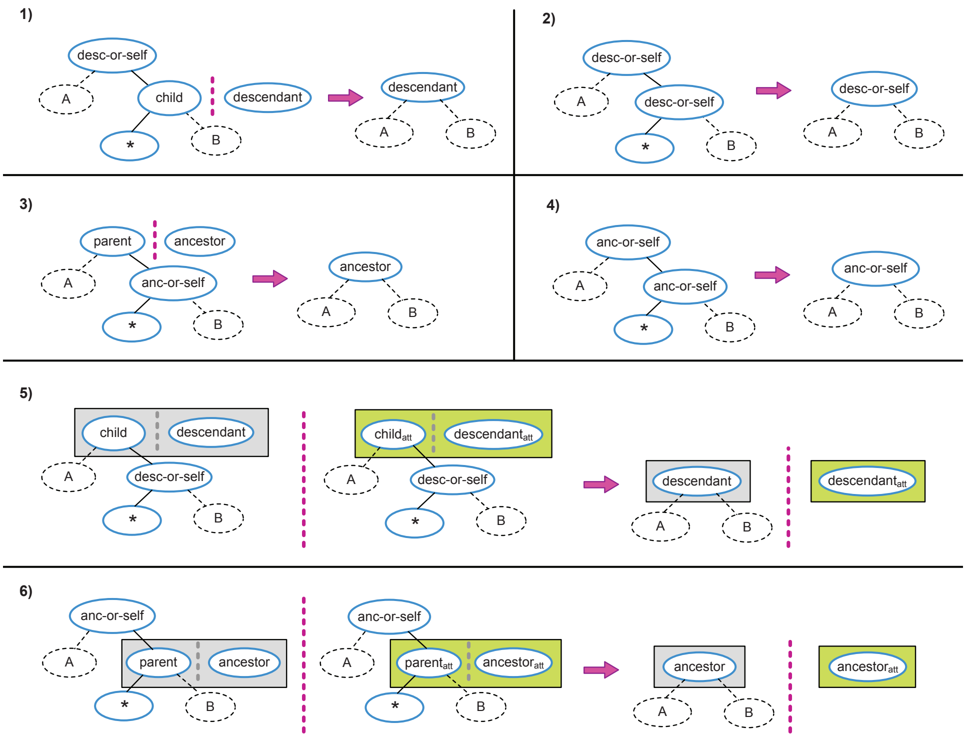


Figure 6.8: Transformations of the *Redundancy suppression* category.

they are just some examples of the several transformations that fall into this category. They are all depicted in Figure 6.8. While preserving the semantics of the original query, this kind of modifications saves intermediate results generation.

(b) *Synonyms translation*: with this modification we aim to replace an axis with another equivalent (that is, producing the same results), and to produce sequences of same steps that could be further optimized in *Steps unification*. Figure 6.9 illustrates these equivalences.

(c) *Steps unification*: this optimization is devoted to reduce the number of steps to be performed, by integrating several identical steps over the wildcard $*$, into a single one. For instance, let us consider the example

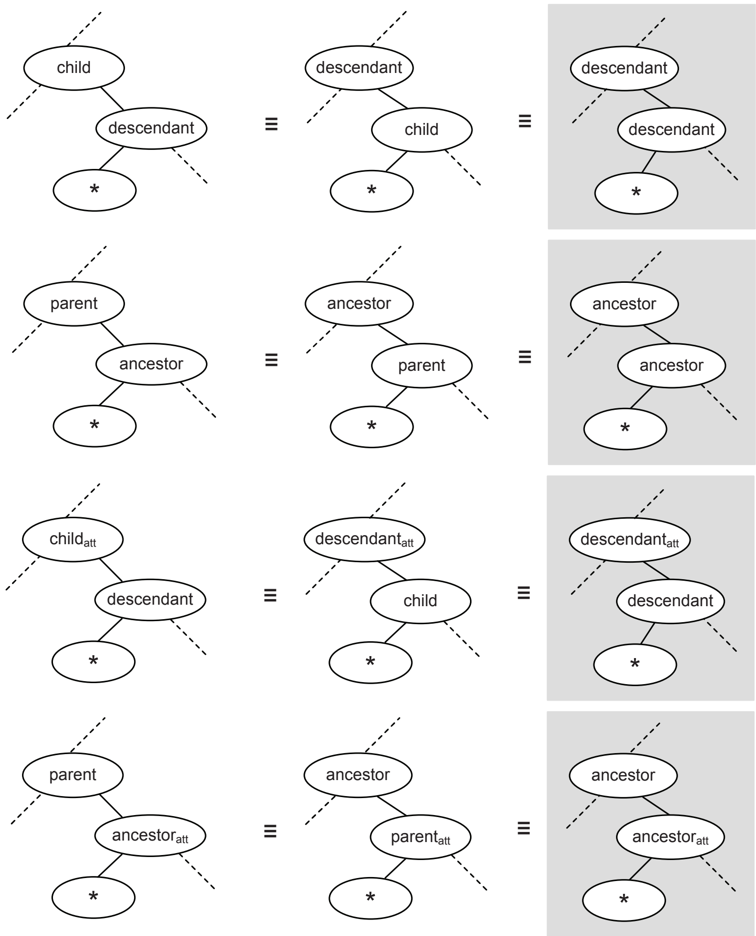


Figure 6.9: Equivalences of the *Synonyms translation* modification.

of Figure 6.10. If we observe the XML document fragment depicted in that figure, we will notice that tags describing the same concept may receive a different name depending on the continent we are considering[5]. Hence, to answer the query posed in that figure we should formulate it as `/world/*/*/*/image`. That is, we are interested in all those `image` elements at distance 4 from the first element of the document, which is `world`. Instead of iteratively cover each `child` step involving wildcards,

[5]Remark `country` tag in case of `europe`, and `region` tag, for `asia`.

we can perform just one step, by creating a new operator, $\mathtt{child}_{\mathtt{dist}_4}$, whose semantics arises from that corresponding to the reduced axis (that is, `child` in this case), but modified to also validate a distance parameter. Figure 6.11 shows some other different scenarios for which this modification applies.

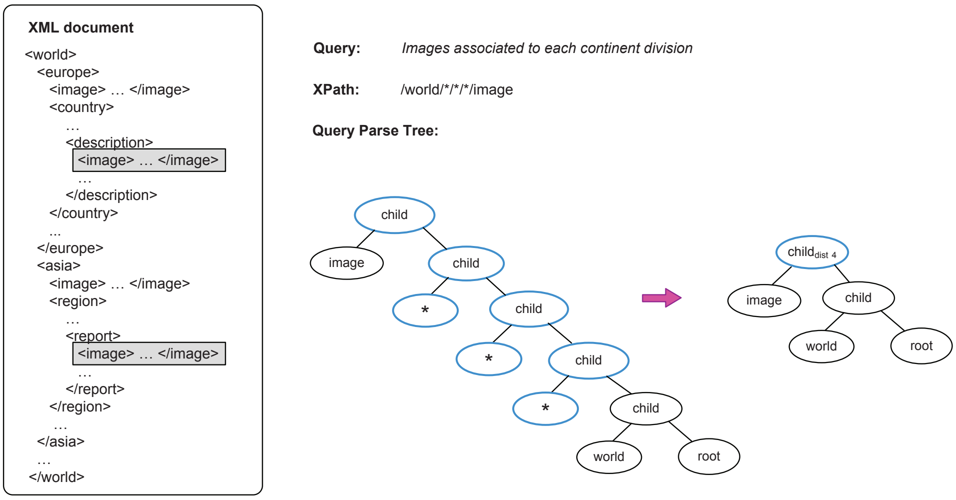


Figure 6.10: Example of *Steps unification*.

3. *Or/and optimizations*: this category regards several modifications that try to simplify the query parse tree taking into account the `or` and `and` logical operators properties. Some of them stem from algebraic simplifications like the following: $(A \cap B) \cup (A \cap C) \equiv A \cap (B \cup C)$, and $(A \cup B) \cap (A \cup C) \equiv A \cup (B \cap C)$. However, we also perform some other transformations such as discovering duplicated tree patterns related through an `or` operator, and flattening `and` operators evaluated over a same element/attribute.

   For instance, Figure 6.12 shows an example of the first scenario. Notice that the first step at both sides of the `or` operator delivers `book` elements that are parents of a valid `chapter`. Hence, it can be set one level up as a common step, while moving downwards the `or` logical operator (see Figure 6.12 b)). The same situation is then encountered, but with respect to `chapter`. So, we proceed in a similar manner (see Figure 6.12 c)).

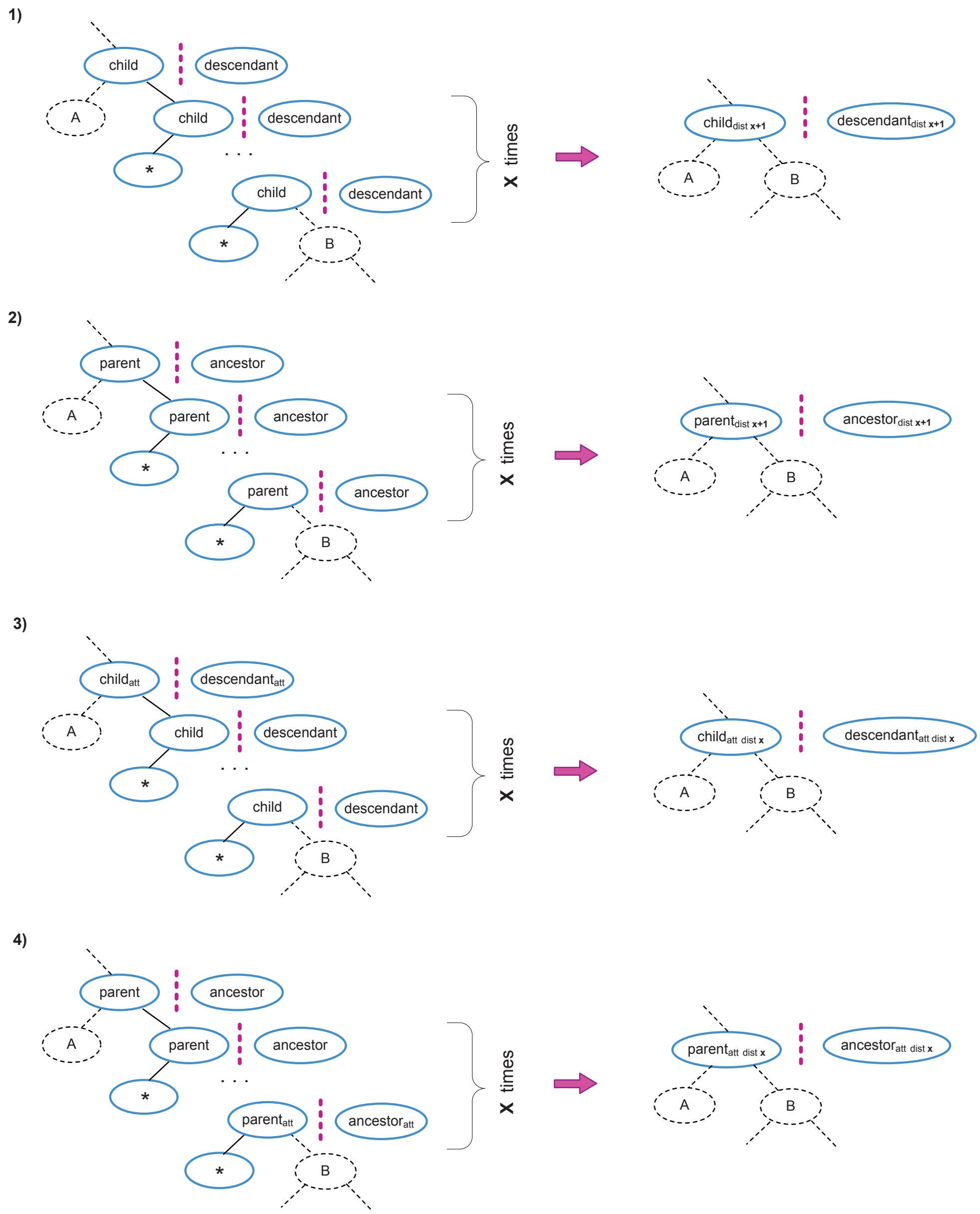


Figure 6.11: Typical scenarios of *Steps unification*.

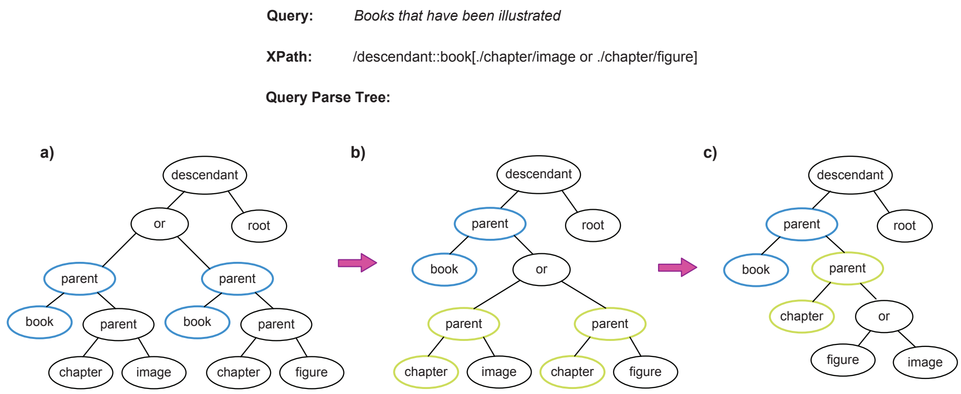


Figure 6.12: Example of *or* optimization.

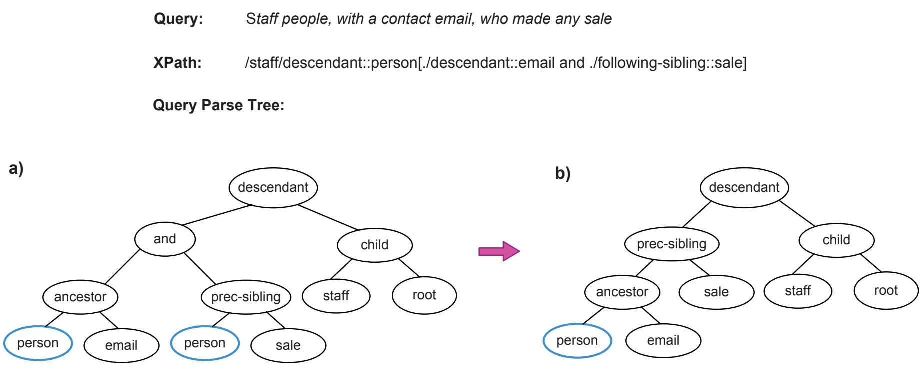


Figure 6.13: Example of *and* optimization.

On the other hand, and regarding the `and` operator, an example of the corresponding transformation is depicted in Figure 6.13. In this case, the `and` operator is relating different steps, namely `ancestor` and `preceding-sibling`, but over instances of a same element node, which is `person`. Since the semantics of the `and` operator implies the fulfillment of the predicate conditions of both sides, that is, the retrieved `person` element node must be an `ancestor` of an `email` element, but also it must precede (as well as be sibling) a `sale` element, both conditions can be composed on a same branch of the query parse tree (see Figure 6.13 b)).

4. *Root node deletion*: it stands for a minor transformation that saves performing an unnecessary validation. Since the root node constitutes the root of the hierarchy, we know that any other element descends from it. Hence, we can omit any step involving a `descendant` selection from the root node. Figure 6.14 illustrates the three different scenarios that belong to this category.

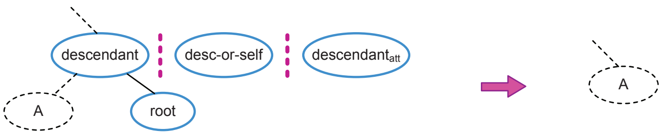


Figure 6.14: Scenarios of *Root node deletion*.

6.4 Final Query Plan: the Query Execution Tree

Previous section described in detail the different modifications we consider[6] before obtaining the final *query execution tree*. We next discuss some important features related to their relevance into the general evaluation process.

As the reader could notice the first two groups of transformations are intended to meet XWT features. For instance, the so called *Attributes equality simplification*, aims to make use of the XWT procedures designed to deal with phrase patterns, instead of having to separately search for the attribute name and its value and then operate with them. In turn, the combination of the three types of *Wildcard optimizations* attempts to reduce as much as possible the number of location steps whose node test is the wildcard '*'[7], and ultimately, also to exploit the XWT ability to obtain the depth of any element/attribute [8]. Remember that, as explained before, most scenarios can be translated into a single step based on a depth (distance) test. Notice, as well, that to reach this final goal, the transformations of this category must be performed in the same order they have been explained in Section 6.3 (that is, 1) *Redundancy suppression*, 2) *Synonyms translation*, 3) *Steps unification*), as they are strongly dependent. On the other hand, both *Or/and optimizations* and *Root*

[6] This work does not focus on full optimization features. This is an issue worth of further work.
[7] Notice that '*' potentially selects all occurrences of any element/attribute, which makes a location step over it be extremely costly.
[8] Thanks to its linkage with the balanced parentheses representation.

node deletion scenarios constitute general transformations that are not specifically intended to benefit from XWT properties. Rather they aim to save processing time during query evaluation, by considering the evaluation strategy we use, which is next explained in Chapter 7.

If we now consider the overall set of modifications as a whole, one can note that there are not tight dependencies as those pointed out inside the *Wildcard optimizations* category. Hence, they are not tied to an specific global order. Yet, we must consider that *Wildcard optimizations* must precede the *Root node deletion*, to determine if the last one applies or not. Figures 6.15 to 6.20 illustrate an example of the global transformation procedure performed to make up the final query execution tree of the following query sample[9]: `/*/descendant- or-self::* /paper[./parent ::journal or ./parent::book] /content/ */*/summary [./ @keyword="XML"]`.

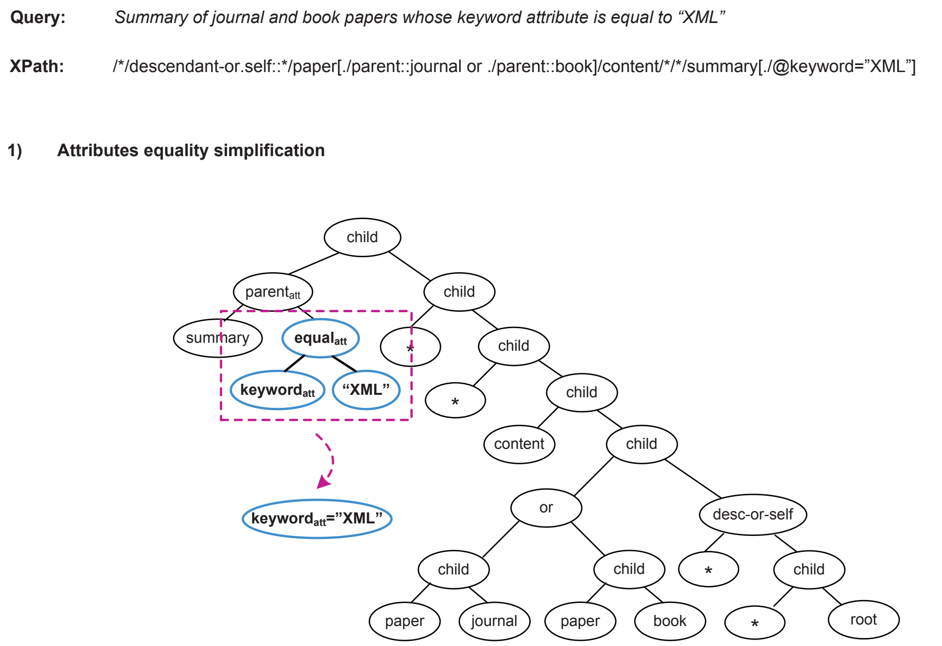


Figure 6.15: Application of *Attributes equality simplification* transformation over the initial query parse tree.

[9]Let us assume an XML document for which such a query applies.

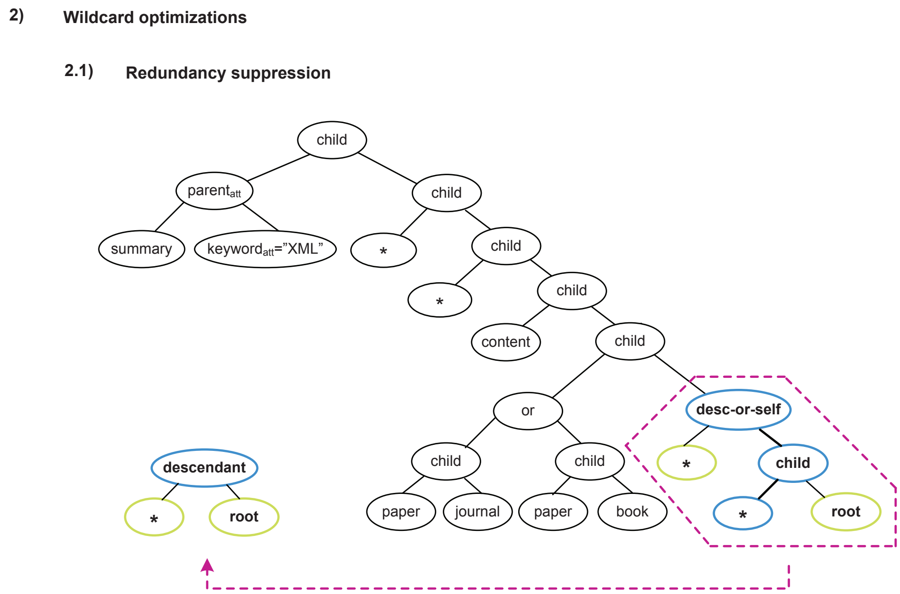


Figure 6.16: Application of *Redundancy suppression* transformations over the query parse tree obtained from Figure 6.15.

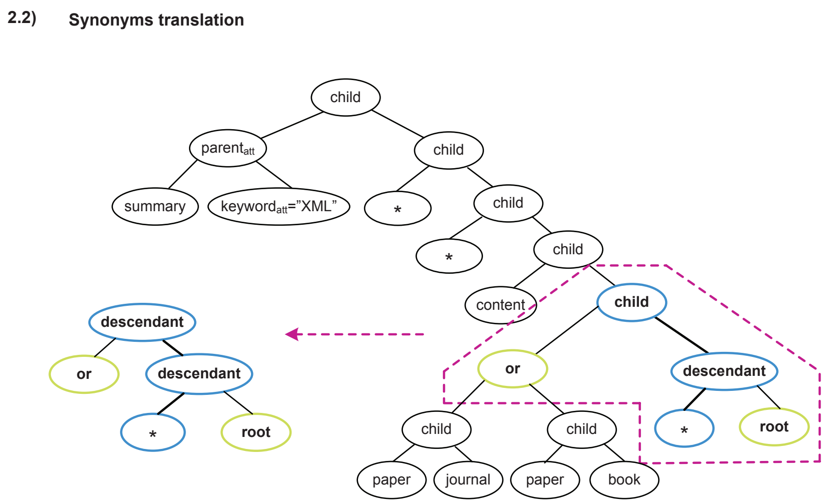


Figure 6.17: Application of *Synonyms translation* modification over the query parse tree resulted from Figure 6.16.

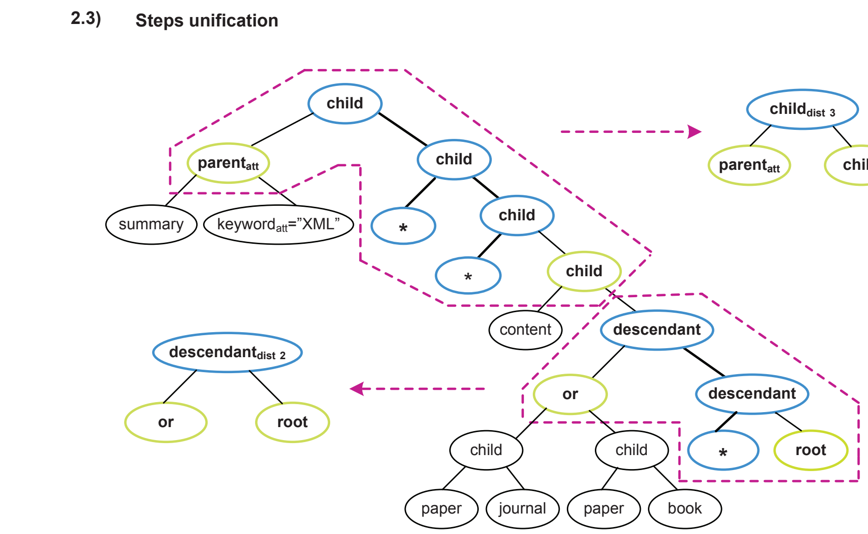


Figure 6.18: *Steps unification* transformations applied over the query parse tree obtained from Figure 6.17.

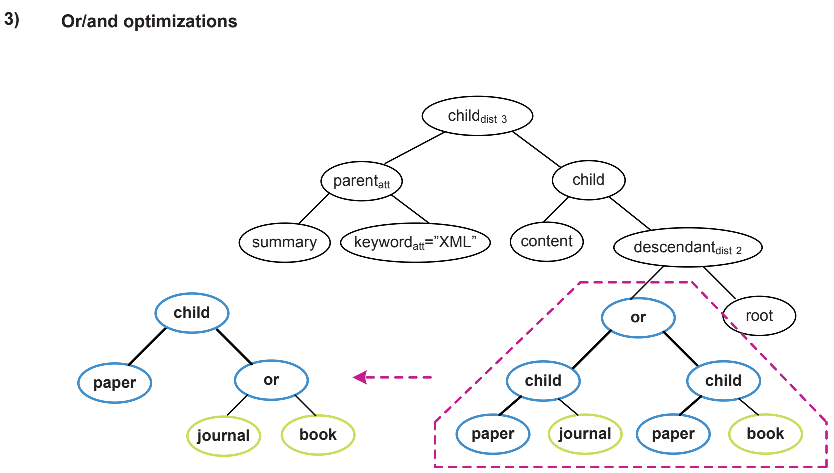


Figure 6.19: *Or/and optimizations* applied over the query parse tree resulted from Figure 6.18.

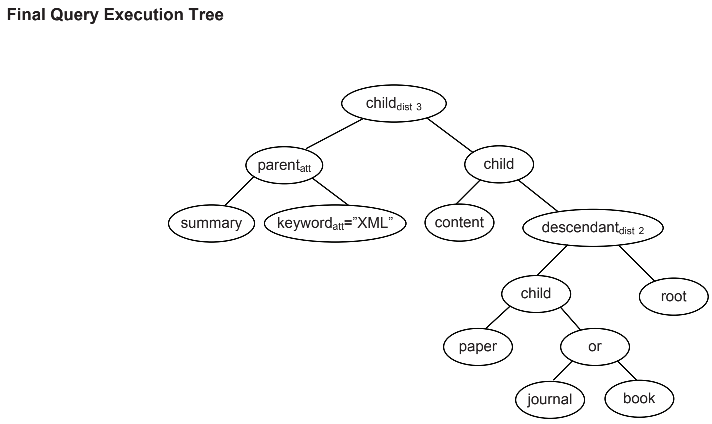


Figure 6.20: Final query execution tree of the query example described in Figure 6.15.

As we reach the final query execution tree (see Figure 6.20), this can be used as input for the next submodule: the *Query evaluator*. Notice that each node of the query execution tree will be directly translated into an operator that stands for the specific component/axis/function that it is representing. Chapter 7 addresses the query evaluation process, given a final query execution tree.

Chapter 7

Query Evaluation

Chapter 6 focused on the *Query parser* component of the *Query module* of XXS, and covered the description of the preliminary query parsing up to its realization as an execution plan, given by the so called *query execution tree*. Now, we regard the second component of the XXS *Query module*, namely the *Query evaluator*, and address the actual evaluation of the final query execution tree obtained from the previous submodule (see Figure 7.1). In this way, Section 7.1 is devoted to provide a conceptual description of the general evaluation procedure that the *Query evaluator* performs, and to discuss the main strategies that characterize it. After that, Section 7.2 deepens the implementation of these general concepts, by explaining the two main operational schemes we distinguish depending on whether leaf or internal nodes (of the query execution tree) are considered.

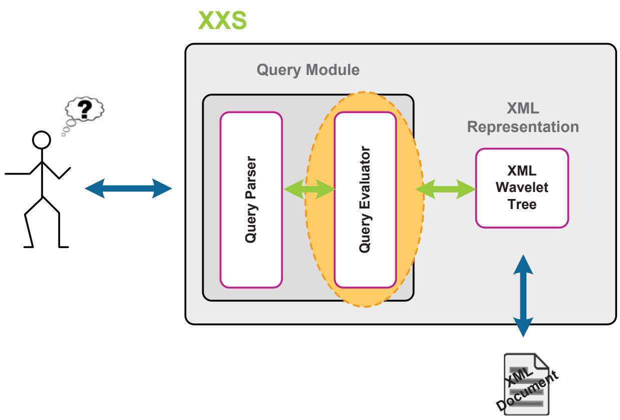


Figure 7.1: *Query evaluator* submodule of the XXS system.

7.1 Conceptual Description

As pointed out in Section 5.4, any component of an XML document (e.g. an element, an attribute, a word, a phrase, etc.) can be ultimately regarded as a segment, $[s, e]$, given by the *start* (s) and *end* positions (e) of the text that the own component covers. Recall, for instance, that in case of an element, the initial and final segment positions arise from that of the corresponding start-tag and end-tag, respectively. Similarly, the segment of a phrase is given by the positions of the first and last word of the pattern. Moreover, any single word (e.g. an attribute name, a word of the textual content, etc.) stands for a particular case of segment starting and finishing at the same position. This common representation constitutes one of the key features of our query evaluation, since, as it will be next described, it is based on the use of segments [NBY95].

Given a query execution tree, the overall evaluation procedure starts by demanding the first result to the root node. This request is sent down through the tree nodes of the query execution tree until reaching the leaves. Note that tree nodes are either leaf nodes or internal nodes.

- **Leaf nodes**: they constitute the basic extraction operands. Each leaf node retrieves, from the XWT, the occurrences (segments) of the specific component that it represents, and returns the valid segment found to the tree node above it.

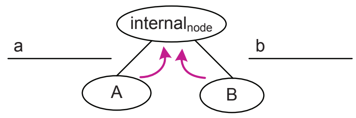
- **Internal nodes**: these are operators that compare the segments they receive from both sides, using the comparison relations between segments shown in Section 5.4[1]. Notice that the internal node semantics (that stems from the semantics of the axis or text function that the own node represents) indicates the type of relationship that the received segments should keep. In Figure 7.2 we show, for the most common XPath axes, the target relation that the received segments must satisfy to meet their semantics. Remark that, in some cases, additional checks would be needed, in addition. That is, for some operators, compared segments not only must keep a given relationship, they also must fulfill, for instance, to have a given depth (see `parent` and `child` axes in Figure 7.2), or even to share a common parent (see `following-sibling` and `preceding-sibling` axes in Figure 7.2)[2]. For example, in Figure 7.3, each time the *contains* node receives an `article` segment, $[a.s, a.e]$, and a **"`Olympic games`"** segment, $[t.s, t.e]$, it must check whether the $\supset$ relation holds, that is, $a.s < t.s$ and $a.e > t.e$.

  If the compared segments satisfy the required relationship, the internal node sends upwards (that is, to its parent node in the query execution tree) the

[1] Recall $<$, $>$, $\subset$, $\supset$, and $=$.

[2] In Chapter 8 a detailed description of the target relationships and additional validations required by all the different operators, apart from those shown in Figure 7.2, is provided.

Internal$_{node}$	Relation
ancestor	a ⊃ b
descendant	a ⊂ b
parent	a ⊃ b [1]
child	a ⊂ b [1]
following	a > b
preceding	a < b
following-sibling	a > b [2]
preceding-sibling	a < b [2]
self	a = b



[1] Additional validation of the segments depth
→ depth(a) = depth(b) – 1 (parent)
→ depth(a) = depth(b) + 1 (child)

[2] Segments must share the same parent, in addition

Figure 7.2: Target relations that compared segments must keep to satisfy the semantics of an internal node representing different XPath axes.

segment received from its *left* child[3]. Otherwise, the internal node will keep searching, consuming results from either child, until if finds a segment from the left side that fulfills the required relationship with a segment of the right side. During this search, the request of new segments from both sides will be based on the result of the comparison between current segments. That is, depending on the relationship that current segments actually keep, and the relationship that they should fulfill to meet the internal node semantics, this node determines the side from which a new result will be required to continue the process.

By following this operational scheme, results retrieved by each leaf or internal node are sent upwards, until the root of the query execution tree, which operates accordingly, finally delivers the first result. At this point, the whole procedure is repeated again searching for the next query result, in such a way that results are retrieved one by one, providing a *lazy evaluation* scheme, in which results can be delivered on user demand.

Example Let us assume the query execution tree of Figure 7.3 to show how our general evaluation scheme works when executing the query `//image [contains (./parent::article, “Olympic games”)]`. As stated, the evaluation always

[3] This is the general behavior, with the exception of the `or` operator, which may deliver segments from both sides.

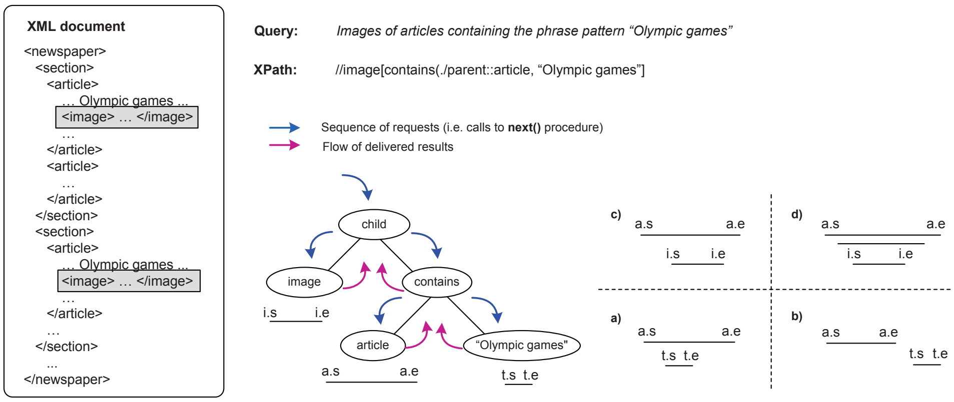


Figure 7.3: General query evaluation scheme.

starts by asking for the first result to the root node of the query execution tree. Since it is an internal node, it must proceed by comparing the segments received from both children (i.e. from both sides). Hence it first propagates the request downwards to obtain those segments. The left side of the root node is a leaf node, therefore this node retrieves the segment associated to the first occurrence of the `image` element, and then delivers it to its parent (the root node, in this case). In turn, its right child is an internal node again (the *contains* one), so it proceeds by demanding to its children the first `article` and "`Olympic games`" segments, respectively, to operate with them. Once the *contains* node receives these segments, it compares them by checking if the `article` segment contains or not the received segment of "`Olympic games`". In the former situation, we have a hit, thus *contains* reports the `article` segment to the node above it, to continue the process in a same way up (see Figure 7.3 a)). Otherwise, and depending on the comparison result, next occurrences of either child of *contains* will be requested, to proceed with comparisons until finding a valid `article` segment (that is, an `article` containing the phrase pattern "`Olympic games`"). For instance, in Figure 7.3 b) we can see that $a.e < t.s$, therefore *contains* would ask for the next `article` occurrence to continue validations. Finally, when *contains* finds a valid `article`, the *child* node of the query execution tree can operate. In case that the received first segment of `image` is a child of the `article` segment delivered by *contains*, then we can produce the first query result (see Figure 7.3 c)). In turn, if both segments do not fulfill the *child* semantics (for example, in Figure 7.3 d), we can see that `image` is a descendant of `article`, but not a direct child, as their depth difference is greater than 1) the process continues with the *child* node requesting the next `image` segment or `article` segment containing "`Olympic games`", accordingly, depending on the relation between the current segments.

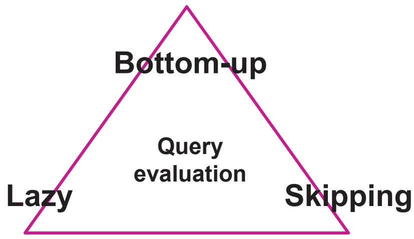


Figure 7.4: Main strategies that characterize XXS query evaluation.

7.1.1 Evaluation Strategies

As stated, the general evaluation scheme combines both a *bottom-up* approach, which starts from the leaf nodes of the query execution tree and works its way up to the root (see flow of pink arrows in the query example of Figure 7.3), and also a *lazy evaluation* plan[4], as final results can be provided by a loop that sequentially obtains them on demand.

Yet, there is still another important factor that determines the efficiency of XXS (see Figure 7.4). Recall that internal nodes keep on requesting segments from either side whenever the current ones do not fulfill the imposed relationship. As stated, the decision of which side it has to ask for a new segment is done depending on the relation that the current segments satisfy and that required according to the node semantics. But what is more important is the fact that the sent request makes use of a *skipping* strategy: the request will be actually restricted by a *minimum admissible position* that the next retrieved segment has to accomplish.

For instance, let us consider the example at the top of Figure 7.5, where `movie` elements that are ancestor of any `rating` element are retrieved. We can see that the current segments (those marked in bold face in the figure), do not fulfill the *ancestor* axis condition, so instead of just requesting the next occurrence of `movie` in a sequential order, we can perform a more intelligent procedure and ask for the next occurrence of `movie` finishing after the end position of `rating` (that is, $m'.e > r.e$). In this way, we avoid visiting all those occurrences of `movie` that could happen before the current occurrence of `rating` and which are not useful.

A similar example, but regarding the *descendant* axis is sketched at the bottom of Figure 7.5. In this case, the current segments do not satisfy the descendant relationship either. What is more, given their current relation, $r.e < m.s$, just an occurrence of `rating` starting after the beginning of the current `movie` segment

[4]This kind of evaluation may (although it does not have to) be slower than a full oriented one, however it is the optimal choice to save space, specially when working with large text databases, since it avoids to store all intermediate results into main memory.

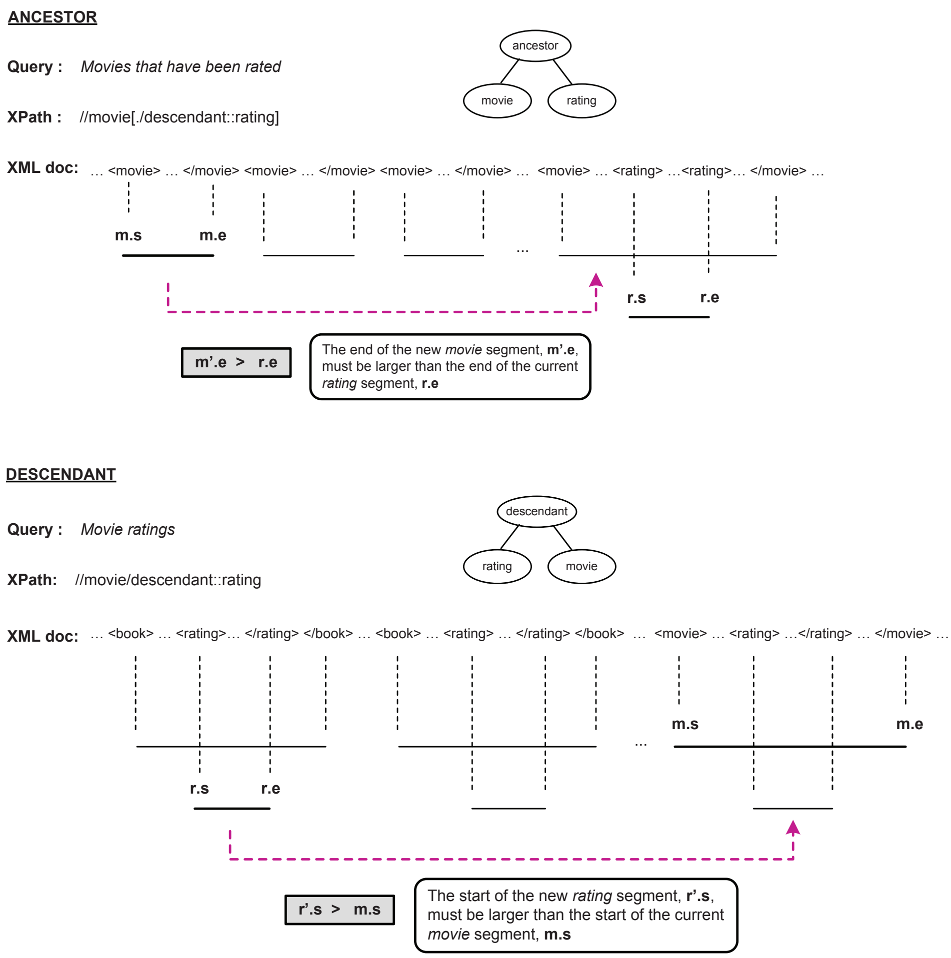


Figure 7.5: Skipping of segments.

could fulfill the desired relationship. Hence, the search for the next valid occurrence of `rating` may be more accurately performed, avoiding also to visit useless `rating` segments, if it regards that condition, that is, if we seek for the next `rating` occurrence fulfilling $r'.s > m.s$.

Therefore, formally, when a node of the query execution tree is required to deliver a new segment, it will perform a *position restricted retrieval* regarding the start or end position of the new requested segment, as applicable. Note that according to this evaluation model segments are traversed in preorder, but only visiting *relevant* ones, that is, segments that we must touch, as a minimum, in order to answer the query. This general behavior is similar to the idea of the *staircase join strategy* [GvKT03] referred in Section 4.1.2.2.

7.2 General Implementations

We have just conceptually described the general query evaluation strategy. In this section, we deepen on its actual implementation. Notice that the evaluation process is ultimately regarded as a sequence of linked requests (see blue arrows of Figure 7.3) demanding new segments to either a leaf or to an internal node, modified by the positional restrictions (that is, the *minimum admissible positions*) that the requested segment must fulfill. In practice, these requests are implemented through a procedure we call *next*. We next discuss the practical details of this procedure, by considering the operational scheme of both leaf and internal nodes, regardless the component/axis/function that they may represent. The implementation of the *next* procedure for each particular component/axis/function will be later analyzed in Chapter 8.

7.2.1 Leaf Nodes

Leaf nodes are in charge of delivering the basic components, that is, elements, attribute names, words and phrase segments. The *next* procedure of a leaf node commonly receives a single positional restriction which is referred to the start position of the segments that it retrieves. Still, in case of elements, it will admit positional restrictions related to the element start-tag and to its end-tag (that is, to the segment start position, but also to its end limit). Note that positional restrictions are generated by *internal* nodes of the query execution tree during query evaluation, and transmitted downwards through requests to its child nodes, which in turn may generate as well other restrictions that ultimately apply over a same *leaf* node. As shown in Figure 7.5, some restrictions generated by *internal* nodes working over elements may refer to the element start-tag, while others will be referred to the element end-tag. Hence, at last, a *leaf* node delivering elements may receive positional restrictions related to each of the element limits[5] . Therefore, in this particular situation, the first step of the *next* procedure will determine which of the two incoming positional restrictions (and thus which of the two element limits, namely the element start-tag or its end-tag) should be used to perform the retrieval. To ensure the best *skipping*, the most forward incoming positional restriction will be always selected.

Then, given a positional restriction, p, the *next* procedure of a leaf node mainly consists of first *counting* the number of occurrences of the specific component that

[5] In Section 7.2.2, where the general *next* procedure of an *internal* node is explained, reader will get further insight into the actual positional restrictions propagation.

the leaf node represents[6], c, until that position, $k = count(c,p)$, and second, *locating* the $(k+1)^{th}$ occurrence of it. Recall that one of the main advantages of the XWT data structure is the implicit self-indexing capabilities it provides, which, precisely, permit to efficiently count the number of occurrences of a word in the document, but also up to a specific position, and to locate any occurrence of a word, as shown in Section 5.2.2. Therefore, these basic operations become key to implement the *next* procedure of a leaf node.

7.2.1.1 Further Discussions

As it will be further explained in Chapter 8, the general behavior of the *next* procedure of a leaf node may be slightly modified in case that *self-nested* XML elements are involved, and also when dealing with phrase patterns.

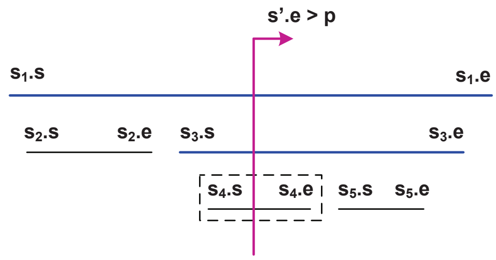


Figure 7.6: Example of *self-nested* elements.

For instance, let us consider the example of Figure 7.6, for a better comprehension of the *self-nesting* scenario. There, we have depicted several `section` segments that exhibit the *self-nested* property. Now let us assume that, in such a scenario, the *next* procedure of the leaf node delivering those `section` segments has to retrieve the next occurrence of `section` whose end-tag ($s'.e$) occurs after the positional restriction set by p (that is, $s'.e > p$). Following the general *next* procedure described above, it would retrieve the segment inside the striped rectangle. Notice that the procedure will be applied over the `section` end-tag, as the restriction is referred to $s'.e$. Therefore, it will start by counting the number of occurrences of a `section` end-tag (represented as $s_i.e$ in Figure 7.6) before p, that is 1 (that corresponding to $s_2.e$), and then it will locate the next one, that is, the 2^{nd} occurrence of a `section` end-tag, that corresponds to $s_4.e$. Hence, in this way it will retrieve the segment s_4. But recall that segments must be delivered in preorder. Thus, all those `section` elements containing the segment s_4, which indeed satisfy

[6]Taking into account that for elements, this procedure may be applied over the element start-tag or over its end-tag, as stated.

the restriction $s'.e > p$, and that appear before s_4 regarding a preorder traversal (namely, s_1 and s_3 segments, that are marked in blue in Figure 7.6), should be previously delivered.

7.2.2 Internal Nodes

Regarding the *internal* nodes, an important feature first to note is that the positional restrictions received by the *next* algorithm always apply to the child node of the query execution tree whose occurrences are delivered by the internal node, that is commonly the node of its left side. Yet in case of the `or` operator, the incoming restrictions may be applied to any of the child nodes, since the delivered results can be obtained from both sides. Notice, as well, that if the *internal* node ultimately delivers any basic component apart from elements, its *next* procedure actually will receive a single positional condition, referred to the start position of the requested segments. Just in case of working over elements, the received conditions will be referred to the start and end limits (that is, to the element start-tag, but also to its end-tag), in line with that mentioned in Section 7.2.1.

Algorithm 7.1: General scheme for the *next* procedure of an internal node

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling the node semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  - Segments comparison:
    $left < right,\ left > right,\ left \subset right, left \supset right,\ left = right$
  - Depending on the comparison result (and eventually, on some additional validations):
  if *we reach a valid result* **then**
    1) We update the left positional restrictions:
      $left_s, left_e$
    2) We deliver the obtained result:
      $result \leftarrow left$; **return** $result$
  else // we retrieve the next $left$ / $right$ valid segment, as applicable
    1) We update the corresponding positional restrictions:
      $left_s, left_e$ / $right_s, right_e$
    2) We request the next segment from the left (L) / right (R) side:
      $left \leftarrow L.next(left_s, left_e)$ / $right \leftarrow R.next(right_s, right_e)$
$result \leftarrow \emptyset$
return $result$

Anyway, those conditions constitute the start point of the *next* procedure of an *internal* node, whose general scheme is sketched in Algorithm 7.1[7]. In this algorithm, we assume the case of an *internal* node representing an operator different from `or` (as delivered segments are obtained from the left side), that operates over elements (as it receives both start and end incoming positional restrictions).

As shown in Algorithm 7.1, the first step of the *next* algorithm of an *internal* node obtains a new segment from the left side (in case of an `or` operator, the new segment will be obtained from the side which delivered the last result), in order to set both sides ready to start the segments comparisons. To this aim a *next* call over the corresponding side is triggered by using the received positional restrictions (see lines from 1 to 3 in Algorithm 7.1). It is important to highlight at this point why this segment request is performed at the beginning of the procedure, and not just after a valid segment is found and sent upwards:

- Whenever a valid segment is recovered by an *internal* node, we could ask for the next segment of the delivered child node, to keep both sides ready for a future *next* request over that *internal* node. However, as stated, we do not do that. Instead, in such a moment, we just proceed by *updating* the positional restrictions that the next requested segment from the retrieved side should satisfy (see lines 11 and 12 in Algorithm 7.1). Note that new received conditions of a future request could imply a better *skipping*. Thus, given that case, if the segment request would has been performed once a valid result is found, we should discard the new segment obtained in the previous call to the *next* procedure and send a new request over the retrieved side with the new better incoming restrictions. In turn, if just an *update* of the restrictions is done after retrieving a valid segment, this avoids doing unnecessary work, by just taking the best of the positional restrictions (see lines 1 and 2 in Algorithm 7.1) between the ones inferred from the previous call to *next* and the new received ones (that is, $left_s$ and $left_e$, and new_s and new_e, respectively, in Algorithm 7.1), and then performing the segment request (see line 3 in Algorithm 7.1).

Once performed the request over the left child (or the corresponding one in case of the `or` operator), the *next* procedure starts by searching for a valid segment (see lines from 5 through 19 in Algorithm 7.1), that is, a segment that satisfies the semantics of the *internal* node. To this aim, the current segments of both sides are compared. According to their relation, and eventually by also verifying some additional checks, we know if a valid segment has been found or not. If the semantics is fulfilled, then the retrieved segment is sent upwards (also updating the related

[7] We denote by L/R the left and right child nodes of the *internal* node, respectively; by *left*/*right*, the cursors to the current segments received from each child node; and we use $left_s$,$left_e$/$right_s$,$right_e$ to represent the positional start and end restrictions that new requested segments from each side must fulfill.

positional restrictions to further obtain a next valid segment, as just mentioned). If not, we determine which of the two current segments must be advanced to keep on searching for a result fulfilling the internal node semantics, and also its *jump*. That is, if we have to ask for a new segment of the left side, then their positional restrictions, namely $left_s$ or $left_e$ in Algorithm 7.1, will be updated accordingly. In case it is the segment of the right side, then $right_s$ or $right_e$ will be rather updated. Finally, the procedure follows by requesting the next segment from either child node, as applicable, and continues the process in the same way.

7.2.2.1 Further Discussions

As new segments are consumed from either child of an *internal* node, until it finds a valid result, positional restrictions are generated. Recall that they depend on the actual relationship kept by current segments, and also on the semantics of the particular *internal* node, that is on the semantics of the axis/function that the *internal* node represents. However, in case of *internal* nodes which work over elements, we also may found that, even for a same axis/function, these conditions are different depending on whether it operates over elements that are *self-nested* or not. As a result, a same *internal* node may finally lead to several implementations of the *next* procedure. Chapter 8 describes in detail these different implementations for each of the axes/functions that an *internal* node may stands for.

Chapter 8

Implementations Description

Chapter 7 introduced a conceptual description of the query evaluation procedure performed by the *Query evaluator* submodule of XXS. It also provided some basic general notions about the practical implementation of that procedure, depending on whether *leaf* or *internal* nodes of the query execution tree were considered. In this Chapter we explain in detail the particular implementations of each component/axis/function that a *leaf* or *internal* node may stand for, according to the subset of XPath addressed in this work. Section 8.1 first details some preliminary remarks about practical segment representations, to consider in the rest of the chapter explanations. Then, Section 8.2 focuses on the description of the different implementations.

8.1 Practical Segment Representation

Prior to starting with the core of this chapter, it is important to highlight a practical consideration that differs from the conceptual description explained in Section 7.1, concerning the segment representation assumed for XML elements. Recall that leaf nodes recover from the XWT the occurrences of elements, attributes, words, etc., represented as segments, $[s, e]$, with start and end positions given by their limits in the text. That is, s and e correspond to positions in the root node of the XWT. However, in case of elements, we also consider an alternative representation where those initial and final positions arise from the positions of the element start-tag and end-tag regarding the structure of the XML document, that is, regarding the *XDTree* node (or analogously, the balanced parentheses structure). Figure 8.1 depicts both assumed representations.

Whenever elements are the components related through an internal node representing an XPath axis, this alternative representation is used for segment comparisons. Note that relations between element segments can be equally

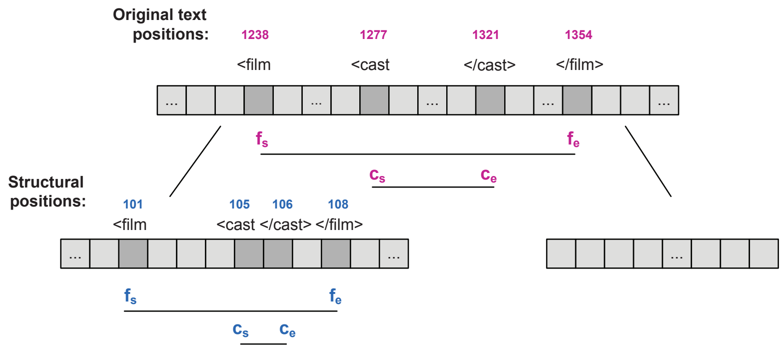


Figure 8.1: Different segment representations for elements.

determined by modeling them in that way, given that the XWT nodes are built by following the order of the words in the text. Yet we obtain a better performance, since we can take advantage of the fact that the *XDTree* node positions match those of the balanced parentheses representation, and also save doing additional *select* operations, as we do not have to go up to the root of the XWT to gather the actual text positions, unless it is needed. Anyway, the regular representation may be used as well for elements, when necessary. For instance, to compare them with respect to other components than elements, for which only the regular representation applies, or for displaying purposes, as well.

8.2 Implementations

In Section 7.2 we showed that the general evaluation procedure could be ultimately regarded as a sequence of linked requests modified by incoming positional restrictions, implemented through the so called *next* procedure. Notice that different general *next* procedures were devised depending on working with *leaf* or *internal* nodes of the query execution tree. We will next describe the practical implementations of these *next* procedures by considering the different components/axes/functions that a *leaf* or *internal* node may represent. Observe that, whenever applicable, pseudocodes will mark in pink color the operations performed over the balanced parentheses data structure, to emphasize the relevance of its use in combination with the XWT structure.

8.2.1 Leaf Nodes

Elements, attributes, words and phrases are the basic components delivered by *leaf* nodes. In case of elements, as previously uncovered in Section 7.2.1, we distinguish three different *next* implementations depending on whether or not the elements are *self-nested*, but also in case that the leaf node represents the wildcard '*'. In turn, attributes and words share the same *next* procedure, as they can be dealt in a same way. Finally, we also distinguish two different scenarios (hence, also two different *next* procedures) when working with phrase patterns, one that supports text searches over phrases spanning more than one text node, and the other which does not.

Algorithm 8.1: *Next* procedure of a non *self-nested* element

```
    Input: new_s, new_e (new positional restrictions)
    Output: next valid occurrence of the element
1.  if new_s ≥ new_e then
2.      occ_s ← count(tag_s, new_s)
3.      if occ_s + 1 ≤ n_tag then
4.          pos_s ← locate(tag_s, occ_s + 1)
5.          pos_e ← findclose(pos_s)
6.          result ← segment(pos_s, pos_e)
7.      else
8.          result ← ø
9.  else
10.     occ_e ← count(tag_e, new_e)
11.     if occ_e + 1 ≤ n_tag then
12.         pos_e ← locate(tag_e, occ_e + 1)
13.         pos_s ← findopen(pos_e)
14.         result ← segment(pos_s, pos_e)
15.     else
16.         result ← ø
17. return result
```

8.2.1.1 Elements

Non self-nested elements. If elements are not *self-nested*, the *next* procedure is rather simple. Algorithm 8.1 shows the pseudocode. It first selects the best incoming positional restriction between new_s and new_e[1], and determines if the search will be performed with respect to the element start-tag or with respect to its end-tag. Then it proceeds with the XWT *count* procedure applied over the element start/end-tag until the new_s/new_e position, obtaining occ_s/occ_e occurrences (see lines 2 and 10 in Algorithm 8.1). If we have not reached the last appearance of the start/end-tag, the algorithm continues by *locating* the $occ_s + 1/occ_e + 1$ occurrence of the corresponding tag. Remember that the XWT *locate* procedure performs

[1]Recall that new_s refers to the element start-tag, while new_e fixes the minimum admissible position for its end-tag.

Algorithm 8.2: *Next* procedure of a *self-nested* element

Input: new_s, new_e (new positional restrictions), $last_s$ (start-tag position of the last delivered segment), $stack$

Output: next valid occurrence of the element

1. **if** $new_s \geq new_e$ **then**
2. $inspectStack(new_s)$
3. $occ_s \leftarrow count(tag_s, new_s)$
4.   **if** $occ_s + 1 \leq n_{tag}$ **then**
5. $pos_s \leftarrow locate(tag_s, occ_s + 1)$
6. $pos_e \leftarrow findclose(pos_s)$
7. $result \leftarrow segment(pos_s, pos_e)$
8.   **else**
9. $result \leftarrow \emptyset$
10. **else**
11. $max_s = max(new_s - 1, last_s)$
12. $inspectStack(new_e, new_s)$
13. $occ_e \leftarrow count(tag_e, new_e)$
14.   **if** $occ_e + 1 \leq n_{tag}$ **then**
15. $pos_e \leftarrow locate(tag_e, occ_e + 1)$
16. $pos_s \leftarrow findopen(pos_e)$
17.   **if** $pos_s \leq max_s$ **then**
18. $new_s \leftarrow pos_e + 1$; *go to* 3.
19.   **else**
20. $result \leftarrow \emptyset$
21.   **return** $result$
22. $occ_s \leftarrow count(tag_s, pos_e)$
23. $occ_{nested} \leftarrow occ_s - occ_e - 1$
24. $match \leftarrow pos_s; i \leftarrow 0$
25.   **while** $i < occ_{nested}$ **do**
26. $parent_s \leftarrow enclose(match)$
27.   **if** $parent_s > max_s$ **then**
28.   **if** $checkTag(parent_s)$ **then**
29. $stack.push(segment(pos_s, pos_e))$
30. $pos_s \leftarrow parent_s$
31. $pos_e \leftarrow findclose(pos_s)$
32. $match \leftarrow parent_s$; $i \leftarrow i + 1$
33.   **else**
34. $match \leftarrow parent_s$
35.   **else**
36.   **break**
37. $result \leftarrow segment(pos_s, pos_e)$
38. **return** $result$

consecutive *select* operations from the leaves up to the root of the XWT. However, as previously stated, in case of elements we stop the process one level before the root node, that is, at the *XDTree* node in this case, and use the segment representation given by the positions of the element boundaries at this level. The just performed *locate* operation provides us the position of only one of those limits, namely pos_s/pos_e, since the search is performed by considering either the element start-tag or its end-tag. But thanks to the connection between the *XDTree* node and the balanced parentheses representation that works on par of that node, we can also find the position of its matching boundary. We simply need to perform a *findclose*/*findopen* operation over the balanced parentheses data structure, to finally obtain the delivered segment: $[pos_s, \mathit{findclose}(pos_s)]/[\mathit{findopen}(pos_e), pos_e]$.

Self-nested elements. With regards to *self-nested* elements we have to manage situations like that exemplified in Section 7.2. Basically, the problem comes from the need of a preorder delivery of the segments when the search is performed with respect to the end-tag of an element that may contain occurrences of the same element inside it. In that case, given a positional restriction, the general procedure locates the most internal segment that fulfills the condition. Nevertheless, it may be still necessary to check their ancestors in order to find occurrences of the same element that should be previously retrieved (as they also satisfy the restriction, but appear before, if we consider a preorder traversal), while keeping stored the internal (and subsequent) ones into a stack, to be delivered in further requests.

The pseudocode for this scenario is presented in Algorithm 8.2. Regardless the *next* procedure is performed related to the start or end position of an element (that is, related to its start-tag or to its end-tag) the algorithm initially inspects the stack of valid segments located in previous requests, but still not delivered, looking for a segment satisfying the appropriate incoming positional restriction. Notice that in case of a search related to the end-tag, the procedure still considers the restriction referred to the start-tag position, as well, to inspect the stack[2]. If found, the segment will be immediately output[3], without further processing. Any other way, different procedures will be applied depending on the case:

1. If the search is performed regarding the start-tag, we proceed analogously as for elements which are not *self-nested* (see lines from 3 to 9 in Algorithm 8.2)
2. If we work with end positions, an additional scan of the ancestors of the first valid segment found (that is, the first element whose end-tag satisfies the new_e incoming positional restriction, and with an appropriate start-tag[4]), may be

[2]Observe that, in this situation, a segment of the stack fulfilling the new_e positional restriction, may not fulfill new_s.

[3]As the stack precisely aims to keep a preorder delivery of segments.

[4]Note again that the start-tag position of the retrieved element can not precede the maximum value between the incoming start positional restriction (new_s) and the start position of the last delivered segment ($last_s$) (see line 17 in Algorithm 8.2).

required in case that other occurrences of the same element contain it. This can be determined by the number of element occurrences whose start-tag precedes the end position of the current segment, but whose end-tag does not (see lines $22 - 23$ in Algorithm 8.2). If applicable (see lines from 24 through 36 in Algorithm 8.2), ancestors will be visited taking advantage of the *enclose* operation provided by the balanced parentheses representation. Whenever an occurrence of the target element is encountered (by codewords comparisons[5]), we push into the stack the current segment, take the just encountered segment as the new current one (since it should be delivered before), and proceed in a same way with the rest of the ancestors. Notice that the procedure finishes either when reaching the number of self-nested occurrences, or if we get an ancestor whose start-tag precedes the position corresponding to the maximum value between new_s, the incoming start positional restriction, and $last_s$, the start position of the last delivered segment (that is, the occurrence of the element delivered in the previous call to the *next* procedure).

Wildcard '*'. There is still another scenario when requesting the next occurrence of an element: the use of the wildcard '*' applied to elements. In this case, we are not asking for the next occurrence of a specific element, any element will be valid. Hence, on the one hand, we know that we must proceed similarly as done for *self-nested* elements, since all element segments will be regarded as occurrences of a same element type. But, on the other hand, another important feature arises: we do not have to use specific start/end-tags as word patterns for the *count* and *locate* procedures. Instead, we can profit from the use of the balanced parentheses data structure, which precisely makes the difference between start/end-tags through the use of opening/closing parentheses. Therefore, it is enough to replace *count*/*locate* operations over the XWT in Algorithm 8.2 by *rank*/*select* operations over the balanced parentheses structure[6]. This shows again the benefits of using on par both structures, the XWT and the balanced parentheses representation.

8.2.1.2 Attributes and Words

The same *next* procedure can be used to obtain both the next occurrence of an attribute name or the next occurrence of a word. Unlike elements, incoming positional restrictions are solely referred to the start positions of the requested segments, and the *self-nested* property does not apply in this scenario. As a result, the algorithm works in the same way as that used for non *self-nested* elements, but without the initial positional conditions comparison. What is more, in this case we

[5]Recall that, as discussed in Section 5.3, the balanced parentheses data structure efficiently provides the position of the parent of an element through the *enclose* operation, and that we can use that information to then apply the XWT *decode* procedure and to discover its codeword.

[6]There is another minor difference in case ancestors are visited, since we can omit the type test, given that all elements are equally considered (see lines 28 to 38 of Algorithm B.1).

Algorithm 8.3: *Next* procedure of attributes and words

Input: new_s(new positional restriction)
Output: next valid occurrence of the attribute/word

$occ \leftarrow count(patt, new_s)$
if $occ + 1 \leq n_{patt}$ **then**
  $pos_s \leftarrow locate(patt, occ + 1)$
  $pos_e \leftarrow pos_s$
  $result \leftarrow segment(pos_s, pos_e)$
else
  $result \leftarrow \emptyset$
return $result$

do not need to find the end position of a retrieved segment, since both the start and end positions are the same for those components. The Algorithm 8.3 shows the pseudocode for this procedure.

Notice that the same algorithm can be used as well when the wildcard '*' is referred to attributes (e.g. //book/@*). This time, the gain is obtained thanks to having reserved a same first byte, say b_x, for the codewords of all the words of the *attributes* vocabulary during the XWT construction. Hence, in this situation, the codeword associated to the word pattern *patt* in Algorithm 8.3 will merely consists of just one byte, that is, b_x.

8.2.1.3 Phrases

When dealing with phrases we distinguish two different scenarios: *i*) to match a continued phrase pattern[7], or *ii*) to match a phrase pattern that may span more than one text node. The first one is used, for instance, for text matches that stem from *Attributes equality simplifications*, as those presented in Section 6.3 (e.g. .../@name[.="New York"]/... → name="New York"). In turn, the second situation arises whenever equal and contains text functions are involved, since, according to their semantics, a match may occur regardless interleaved start/end-tags, and even processing instructions or comments appear. Let us assume the query //section[contains(.,"trip to Manhattan dreams")]. If an XML document contains the following section fragment: ... <section> ... trip to <keyword>Manhattan</keyword> dreams ... </section> ..., it should be delivered. Hence, keyword start-tag and end-tag must be skipped.

To efficiently perform the *next* procedure in both scenarios, we use the same strategy as that performed in the XWT basic procedures over phrase patterns (see Section 5.2.2.2). That is, the search is focused on the least frequent word of the pattern, and only further validations are done whenever the first bytes of the codewords of each word of the pattern match the previous and next bytes of the

[7] We denote with this terminology a phrase that must appear exactly as shown in the pattern.

Algorithm 8.4: *Next* procedure of a continued phrase

Input: new_s(new positional restriction)
Output: next valid occurrence of the phrase

$new_s \leftarrow new_s + order_{min}$ // $order_{min}$: least frequent word position
$occ \leftarrow count(word_{min}, new_s)$ // $word_{min}$: least frequent word
$i \leftarrow occ + 1$
while $i \leq n_{min}$ **do**
  $pos_{min} \leftarrow locate(word_{min}, i)$
  // *matchPhrase* : tries to match the first bytes of the codewords. Then, if
  // applicable, it continues validating the rest ones
  if $matchPhrase(pos_{min}, phrase)$ **then**
    $pos_s \leftarrow pos_{min} - order_{min}$
    $pos_e \leftarrow pos_s + phrase_{nwords} - 1$
    $result \leftarrow segment(pos_s, pos_e)$
    return *result*
  else
    $i \leftarrow i + 1$
$result \leftarrow \emptyset$
return *result*

Algorithm 8.5: *Next* procedure of an interleaved phrase

Input: new_s(new positional restriction)
Output: next valid occurrence of the phrase

$new_s \leftarrow new_s + order_{min}$ // $order_{min}$: least frequent word position
$occ \leftarrow count(word_{min}, new_s)$ // $word_{min}$: least frequent word
$i \leftarrow occ + 1$
while $(i \leq n_{min})$ **do**
  $pos_{min} \leftarrow locate(word_{min}, i)$
  // *matchSkippedPhrase* : tries to match the first bytes of the codewords,
  // while `skipping` occurrences of start/end-tags, comments and processing
  // instructions. Then, if applicable, it continues validating the rest ones.
  // pos_s and pos_e are discovered during the process
  if $matchSkippedPhrase(pos_{min}, phrase, pos_s, pos_e)$ **then**
    $result \leftarrow segment(pos_s, pos_e)$
    return *result*
  else
    $i \leftarrow i + 1$
$result \leftarrow \emptyset$
return *result*

root node, from the position of the just located occurrence of the least frequent word. Hence, the general scheme of the *next* algorithm presented for single words and attributes can be used now, as well, but regarding the least frequent word of the phrase pattern. Yet we need to include an additional check for a complete phrase matching in the word surroundings once it has been located (see line 6 in Algorithm 8.4 and Algorithm 8.5).

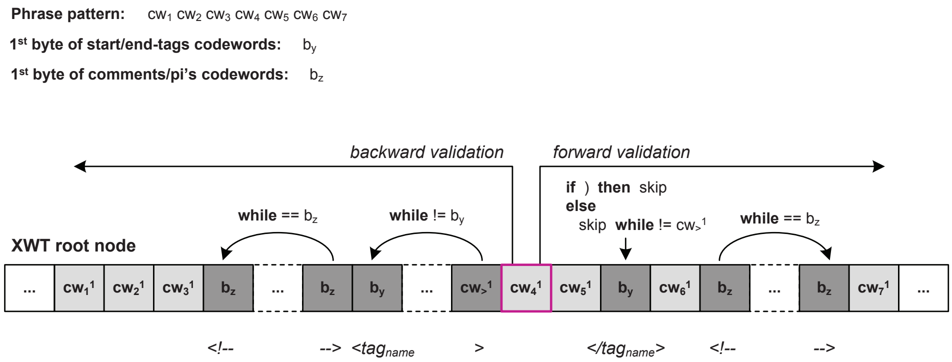


Figure 8.2: First bytes validation with skipping, used to match a phrase pattern.

In the first scenario (that is, when we are searching for a continued phrase), the additional check does not differ from that described for XWT *locate* and *count* procedures over phrase patterns. However, in the second situation (that is, when requesting phrases that may span more than one text node), we also need to skip interleaved occurrences of start/end-tags, comments and processing instructions. Recall that we reserved specific first bytes to code the words of those *special* vocabularies when we assigned codewords during the XWT construction. In particular, here we are interested in the *tags* and *nsearch* vocabularies, which are precisely those whose occurrences we need to disregard. Therefore, the fragments that should be omitted can be easily recognized while first bytes validation is performed in the root of the XWT. Notice that, by doing this, we keep the aim of avoiding further processing, unless the first bytes comparison applies. Figure 8.2 shows how the skipping is performed. In the example, we have considered a phrase composed of 7 words, coded as $cw_1, cw_2 \ldots cw_7$, respectively, and whose first bytes are denoted as $cw_1^1, cw_2^1 ... cw_7^1$. We have also assumed byte b_y to be the first byte of the codewords assigned to start/end-tags, and byte b_z, for comments together with processing instructions, and we have represented the codeword of the right angle bracket, >, as $cw_> = cw_>^1$[8]. Now, let us take the fourth word of the pattern as its least frequent word. Then, we have to perform a backward and forward validation from its position trying to match the first bytes of the codewords of each word of the phrase.

[8]Given its high frequency, > will always be assigned a codeword of only one byte.

In both cases, and regarding comments and processing instructions, occurrences of byte b_z are used as boundaries to skip regions of bytes regardless their values. Yet, for start/end-tags, we follow different approaches depending on the validation direction. Recall that the codewords of any start/end-tag share the same first byte, b_y, and also that a start-tag codeword is always followed by a `>` codeword (although it has not have to immediately appear, since the start-tag may contain any attribute). Hence, if we are moving *backward*, start-tags (and also their attributes) can be skipped whenever we match an occurrence of byte $cw^1_>$, by just omitting byte values until we reach an instance of b_y. Indeed, isolated occurrences of byte b_y (that is, which are not preceded by $cw^1_>$) are also skipped, since we know they are representing end-tags.

In turn, if we are moving *forward*, occurrences of byte b_y may stand for start-tags or end-tags. In case of start-tags, bytes should be skipped until byte $cw^1_>$ is found. However, for end-tags, we just need to disregard that byte and keep on validating the next one. Again, the use of the balanced parentheses data structure is key to discern between both situations. Let us consider that byte b_y is placed at position pos_{b_y} in the root node of the XWT. We only have to compute $count(b_y, pos_{b_y}) = k$ and to inspect the k^{th} position of the balanced parentheses representation, to discover if it matches an opening or closing parenthesis[9]. Then, we can operate accordingly.

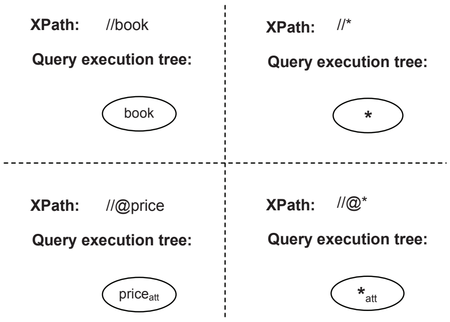


Figure 8.3: Examples to which optimized *next* procedures can be applied.

8.2.1.4 Optimized Leaf Nodes

There are specific queries searching for all the occurrences of an element (e.g. `//book`) (or equally, any element, `//*`), as well as, for all the appearances of an attribute (e.g. `//@price`) (or equally, any attribute, `//@*`), whose final query execution trees are just given by a leaf node representing the element/attribute we are interested in (see some examples in Figure 8.3). In those cases, the query

[9] And by extension, an start-tag or an end-tag, respectively.

Algorithm 8.6: Optimized *next* procedure of specific elements (regardless they are or not *self-nested*)

Input: *order*
Output: $order^{th}$ occurrence of the element

if $order \leq n_{tag}$ **then**
  $pos_s \leftarrow locate(tags_s, order)$
  $pos_e \leftarrow findclose(pos_s)$
  $result \leftarrow segment(pos_s, pos_e)$
else
  $result \leftarrow \emptyset$
return *result*

evaluation can be performed more efficiently by using optimized versions of the corresponding *next* procedures. Since all the occurrences are valid, we can omit the *count* operation related to the incoming positional restriction. We know that the result of this *count* operation will be the j^{th} occurrence delivered by the previous call to the *next* procedure. Hence, just the order of the next occurrence to be requested is needed. Algorithms 8.6, 8.7 and 8.8 present the pseudocode of the procedures for these situations.

Algorithm 8.7: Optimized *next* procedure of *any* element

Input: *order*
Output: $order^{th}$ occurrence of an element

if $order \leq n_{(}$ **then**
  $pos_s \leftarrow select_{(}\ (order)$
  $pos_e \leftarrow findclose(pos_s)$
  $result \leftarrow segment(pos_s, pos_e)$
else
  $result \leftarrow \emptyset$
return *result*

Algorithm 8.8: Optimized *next* procedure of attributes and words

Input: *order*
Output: $order^{th}$ occurrence of the attribute/word

if $order \leq n_{patt}$ **then**
  $pos_s \leftarrow locate(patt, order)$
  $pos_e \leftarrow pos_s$
  $result \leftarrow segment(pos_s, pos_e)$
else
  $result \leftarrow \emptyset$
return *result*

8.2.1.5 Further Discussions

In previous sections, we have just discussed the *next* procedures used when leaf nodes are requested to deliver the next valid occurrence of the basic components (i.e. elements, attributes, words and phrases) they may represent. As one could have noticed, the traversed XWT nodes for each specific pattern over which *count* and *locate* procedures are performed, inside a *next* algorithm, are always the same. Indeed, they will be forward processed. Therefore, *rank* and *select* operations underneath can be sped up by keeping the values obtained from previous calls. That is, by storing the number of occurrences of a byte up to a given position, or by just using pointers to already located occurrences, respectively.

Since several *rank*/*select* operations will be performed for the same byte values, in case of *rank*, the information stored can be used when the target position of the previous rank operation corresponds to the same block as the new sought one, while for *select*, the same applies, but regarding the position of the byte value occurrence previously selected. Hence, instead of counting/searching from the first position of the block, we can start the sequential count/search from the position related to the previous *rank*/*select* operation.

8.2.2 Internal Nodes

The internal nodes of a query execution tree may stand for any XPath axis, but also they may represent the `equal` and `contains` text functions. What is more, remember that also different new axes were devised as a result of the *query parse tree* modifications. For example, it is the case of the axes obtained from the *Steps unification* transformation (e.g. $\texttt{child}_{\texttt{dist}}$, $\texttt{parent}_{\texttt{dist}}$, $\texttt{descendant}_{\texttt{dist}}$, etc.) as well as those related with the use of attributes (e.g. $\texttt{parent}_{\texttt{att}}$, $\texttt{descendant}_{\texttt{att}}$, $\texttt{ancestor}_{\texttt{att\_dist}}$., etc.). We denote all of them as *operators*, for simplicity.

Notice also that, similarly to what happened to leaf nodes delivering element segments, the *next* procedure of those internal nodes which also retrieve element segments, but even of those which do not deliver them at last, but work over elements[10], may result into several versions according to the elements *self-nested* nature. For instance, in case of internal nodes that ultimately receive element segments from both child nodes[11], they lead to four different variants of the *next* procedure, depending on which side exhibits the property:

- *Non-nested*: if none of the elements recovered from the child nodes may contain occurrences of the same element.
- *Full-nested*: if elements from both sides are *self-nested*.

[10] For example, $\texttt{child}_{\texttt{att}}$, delivers attribute segments (left side), but operates over elements, as well (right side).

[11] Remark that each node of the query execution tree ultimately works over segments of any of the basic components: elements, attributes, words or phrases.

- *Left-nested*: if we can find *self-nested* occurrences of elements that come from the left side.
- *Right-nested*: if elements delivered by the right side may contain occurrences of the same element inside it.

These four versions become two in case of internal nodes for which only one of their sides delivers element segments. In that situation we just discern between *non-nested* and *full-nested* variants[12]. Since there is little point in detailing the features of each version for all the operators, we will focus on the performance of the *next* procedure for *non-nested* and *full-nested* scenarios, in order to exemplify both the simplest and more complex variant.

Descriptions will show the most relevant features of each operator regarding the general implementation scheme of the *next* procedure of an internal node, presented in Section 7.2. Therefore, likewise, pseudocodes will equally denote as L/R, the left and right child nodes, respectively, of the internal node in the query execution tree[13]; while they use *left*/*right* to represent the cursors to the current segments obtained from each side. Moreover, $left_s$,$left_e$/$right_s$,$right_e$ will describe the positional restrictions (that is, the minimum admissible start and end positions) that new requested segments from the left and right nodes, respectively, must satisfy.

Algorithm 8.9: *Next* procedure of `ancestor` operator (*non-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `ancestor` semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
    case $left < right$
        $left_e \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
    case $left > right$
        $right_s \leftarrow left.s + 1$; $right \leftarrow R.next(right_s, right_e)$
    case $left \subseteq right$
        $left_s \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
    otherwise
        $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
$result \leftarrow \emptyset$
return $result$

[12] By considering just the side that delivers elements.

[13] Recall that L always represent the side whose segments are sent upwards by the internal node, with the exception of the `or` operator.

8.2.2.1 Ancestor (or-self)

`Ancestor` axis provides a simple example to begin with operators description. Regardless we are in *non-nested* or *full-nested* scenario, the leftmost segment is advanced as long as current *left* and *right* segments are disjoint. Yet the particular advance differs between both situations. In case `left<right`, *non-nested* variant advances to the next left segment which finishes after the end position of *right*. Instead, *full-nested* version moves to the next left segment whose end occurs after the start position of the current right segment, as some nested occurrences of R fulfilling the condition may be contained into *right* (see Figure 8.4 a)). In turn, if `left>right`, both variants move to the next right segment starting after *left* beginning.

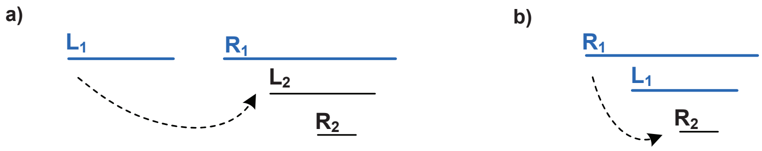


Figure 8.4: Segment advance for `left<right` (a) and `left⊆right` (b) in *full-nested* scenario of `ancestor` axis.

Algorithm 8.10: *Next* procedure of `ancestor` operator (*full-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `ancestor` semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_e \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left \supset right$
    $left_e \leftarrow right.s + 1$; $result \leftarrow left$; **return** $result$
  otherwise
    $right_s \leftarrow left.s + 1$; $right \leftarrow R.next(right_s, right_e)$
$result \leftarrow \emptyset$
return $result$

At some moment, segments are contained one into the other. If it is *left* the contained one (that is, `left⊂right`), *non-nested* alternative advances up to the occurrence of L appearing after the end of *right*. However, *full-nested* option still may find a valid occurrence of R under *left*, that makes *left* become a valid result, hence this time right segment is advanced after *left* start (see Figure 8.4 b)). Finally, if *right* is inside *left* (that is, `left⊃right`), then *left* is sent upwards in both scenarios, first updating $left_s$ or $left_e$, as required.

Algorithms 8.9 and 8.10 give the pseudocode of both *non-nested* and *full-nested* variants. If we replace $\subseteq$ by $\subset$ and $\supset$ by $\supseteq$, we obtain the same versions, but for `ancestor-or-self` axis.

Algorithm 8.11: *Next* procedure of `descendant` operator (*non-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `descendant` semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
    case $left < right$
        $left_s \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
    case $left > right$
        $right_e \leftarrow left.e + 1$; $right \leftarrow R.next(right_s, right_e)$
    case $left \subset right$
        $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
    otherwise
        $right_s \leftarrow left.e + 1$; $right \leftarrow R.next(right_s, right_e)$
$result \leftarrow \emptyset$
return $result$

8.2.2.2 Descendant (or-self)

`Descendant` and `ancestor` are axes with opposite meaning and, thus, opposite behavior under the same cases of segment relations. Again the leftmost segment is advanced whereas current segments are not intersected. Yet, in case of `left<right`, we move to the next left segment beginning after *right* start, regardless the variant. Notice that this behavior is similar to that performed by `ancestor` in the opposite situation, that is, when `left>right`. The same happens to `left>right`, this time with regards to the `ancestor` performance for `left<right`. In this situation, *non-nested* version advances the right segment up to the next occurrence that finishes after the end position of *left*, while the same movement, but referred to the start

position of *left*, is performed for the *full-nested* variant, since nested occurrences of L satisfying the operator semantics could occur inside *left*.

Algorithm 8.12: *Next* procedure of `descendant` operator (*full-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `descendant` semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left > right$
    $right_e \leftarrow left.s + 1$; $right \leftarrow R.next(right_s, right_e)$
  case $left \subset right$
    $left_s \leftarrow left.s + 1$; $result \leftarrow left$; **return** $result$
  otherwise
    $left_s \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return $result$

Once we find no disjoint segments, we can discriminate between `left⊂right` and `left⊇right`. If the first relation applies, then `descendant` relationship is fulfilled, and *left* is delivered upwards. Also $left_s$ is updated to the *left* end position if we are working with *non-nested* elements. Otherwise, it is updated to the *left* start position. On the other hand, the second comparison (that is, `left⊇right`) leads to the search of the next valid segment from the right side beginning after the end of *left*, in case of *non-nested* scenario. Instead, *full-nested* variant still tries to find a valid occurrence of L under *right*, by using its start position as the skipping condition, given that L admits self-nested occurrences. Note again that the axis performance for each of these relations is analogous to the behavior discussed for the opposite ones in the `ancestor` axis. Algorithms 8.11 and 8.12 show the pseudocodes. The operational scheme of the `descendant-or-self` axis can be obtained by doing the same replacement over the `descendant` schema than that pointed out for `ancestor-or-self`.

8.2.2.3 Parent

One can assume that `parent` axis works similarly to `ancestor`, with the proviso that only the segments whose depth level differs in one unit from that of the descendant target segment are valid. However, this solely applies if we deal with elements that are not self-nested. Hence, in that case, the same *next* procedure as that described in Section 8.2.2.1 for the *non-nested* variant of `ancestor` can be used, but including the

Algorithm 8.13: *Next* procedure of `parent` operator (*non-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `parent` semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_e \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $right_s \leftarrow left.s + 1$; $right \leftarrow R.next(right_s, right_e)$
  case $left \subseteq right$
    $left_s \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
  otherwise
    if $depth(right.s) = (depth(left.s) + 1)$ **then**
      $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
    else
      $right_s \leftarrow right.e + 1$; $right \leftarrow R.next(right_s, right_e)$
$result \leftarrow \emptyset$
return $result$

aforementioned validation into the group of actions performed under `left⊃right` comparison. Algorithm 8.13 presents the pseudocode. Note that to check the depth of an element we just need to make use of the *depth* operation provided by the balanced parentheses representation of the XML document structure.

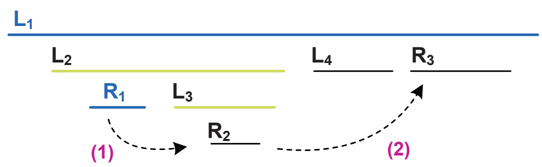


Figure 8.5: Example for *full-nested* variant of `parent` axis.

Any *next* procedure of an internal node does not discard a segment without first ensuring that it has no choice to become a valid result according to the operator semantics. This feature yields some problems in case of the *full-nested* variant of `parent`. Let us consider the example of Figure 8.5, where L_1 and R_1 (see segments marked in blue in Figure 8.5) represent the current segments obtained from the left and right side, respectively. Notice that we may need to traverse all occurrences

Algorithm 8.14: *Next* procedure of `parent` operator (*full-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `parent` semantics

```
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
right ← R.result
while left ≠ ∅ and right ≠ ∅ do
    if getbit(bitmap, left.s) then
        left_s ← left.s + 1; result ← left; return result
    else
        case left < right
            left_s ← left.s + 1; left ← L.next(left_s, left_e)
        case left ⊇ right
            right_s ← right.s + 1; right ← R.next(right_s, right_e)
            if right ≠ ∅ then
                pos_s_parent ← enclose(right.s)
                max_s_parent ← max(max_s_parent, pos_s_parent)
                setbit(bitmap, pos_s_parent, 1)
        otherwise
            right_s ← left.s + 1; right ← R.next(right_s, right_e)
            if right ≠ ∅ then
                pos_s_parent ← enclose(right.s)
                max_s_parent ← max(max_s_parent, pos_s_parent)
                setbit(bitmap, pos_s_parent, 1)
while left ≠ ∅ and left.s ≤ max_s_parent do
    if getbit(bitmap, left.s) then
        left_s ← left.s + 1; result ← left; return result
    else
        left_s ← left.s + 1; left ← L.next(left_s, left_e)
result ← ∅
return result
```

of R that descend from L_1 before finding one that selects it (see the sequence of movements tracked by the striped arrows in Figure 8.5). However, since nested occurrences of same elements could exist, traversed segments of R may be necessary to select further occurrences of L. For instance, in Figure 8.5, the occurrences of R (R_1 and R_2) visited before locating R_3, which qualifies L_1 segment, would select the occurrences of L depicted in green (L_2 and L_3, respectively). Therefore, we need to *remember* traversed R segments to make further occurrences of L qualify. To this aim, an additional bitmap suffices to implement the *full-nested* variant of the `parent` operator. Each traversed occurrence of R flags the position of its parent.

These positions are then checked by left segments to know whether they must be delivered or not (see line 6 in Algorithm 8.14). The pseudocode is described in Algorithm 8.14. Observe that even if the last occurrence of R has been reached, left segments can still qualify (see lines 19 to 23 in Algorithm 8.14).

Algorithm 8.15: *Next* procedure of `child` operator (*non-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `child` semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_s \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $right_e \leftarrow left.e + 1$; $right \leftarrow R.next(right_s, right_e)$
  case $left \subset right$
    if $depth(left.s) = (depth(right.s) + 1)$ **then**
      $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
    else
      $left_s \leftarrow left.e + 1$; $left \leftarrow L.next(left_s, left_e)$
  otherwise
    $right_s \leftarrow left.e + 1$; $right \leftarrow R.next(right_s, right_e)$
$result \leftarrow \emptyset$
return $result$

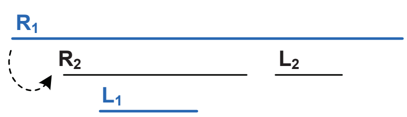


Figure 8.6: Example for the *full-nested* variant of `child` axis.

8.2.2.4 Child

The same note made to `parent` axis regarding the *non-nested* variant can also be applied to `child`. That is, the *next* procedure of the same `descendant` variant can be used, and just to introduce in addition a depth validation in case of `left⊂right` (see Algorithm 8.15). Yet for *full-nested* version, we must follow a different approach. The problem is that given current occurrences from the left and

Algorithm 8.16: *Next* procedure of `child` operator (*full-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `child` semantics

```
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
right ← R.result
while left ≠ ø and right ≠ ø do
    while left ⊈ top(stack) do // stack update
        pop(stack)
    if enclose(left.s) = top(stack).s then
        left_s ← left.s + 1; result ← left; return result
    else
        case left < right
            if isEmpty(stack) then
                left_s ← right.s + 1; left ← L.next(left_s, left_e)
            else
                left_s ← left.s + 1; left ← L.next(left_s, left_e)
        case left > right
            right_e ← left.s + 1; right ← R.next(right_s, right_e)
        case left ⊂ right
            push(stack, right)
            right_s ← right.s + 1; right ← R.next(right_s, right_e)
        otherwise
            left_s ← right.s + 1; left ← L.next(left_s, left_e)
while left ≠ ø and ! isEmpty(stack) do
    while left ⊈ top(stack) do // stack update
        pop(stack)
    if enclose(left.s) = top(stack).s then
        left_s ← left.s + 1; result ← left; return result
    else
        left_s ← left.s + 1; left ← L.next(left_s, left_e)
result ← ø
return result
```

right side, say L_x and R_y, respectively, such that $L_x \subset R_y$, we may have to enter inside R_y in order to find the parent of L_x (or even of some other nested occurrences of L_x). However, later it may occur that R_y qualifies a subsequent occurrence of L_x. Figure 8.5 illustrates an example for better understanding. If we start at L_1 and R_1, we must move to R_2 to properly select L_1. Nevertheless, when we advance to L_2,

R_1 will have been already traversed, and hence L_2 could not be delivered[14]. This situation is managed with the use of a stack of ancestors of the current occurrence of R, which have already been visited. We also keep the invariant *left*$\subseteq$ *top(stack)*. Algorithm 8.16 shows the pseudocode in that case.

8.2.2.5 Parameterized Operators: the *distance* parameter

Once seen the different scenarios faced by `parent` and `child` axes, we are ready to follow with some of the special operators created from the integration of several steps of the *query parse tree* (e.g. $\texttt{parent}_{\texttt{dist}}$, $\texttt{ancestor}_{\texttt{dist}}$, $\texttt{child}_{\texttt{dist}}$, $\texttt{descendant}_{\texttt{dist}}$), as they also meet the same problems. We assume an input *distance* parameter d for all of them. We just refer the main features of each one related to previous procedures.

Parent$_{\mathbf{dist}}$

This operator denotes the selection of an element ancestor which is precisely d levels above the target element. Therefore, *non-nested* variant can be deduced from the same variant of `parent` axis, with the solely modification, in case of `left`$\supset$`right`, of a depth validation that now considers the distance value d (see line 13 of Algorithm 8.13): $depth(right.s) = (depth(left.s) + 1) \Longrightarrow depth(right.s) = (depth(left.s) + d)$. In turn, *full-nested* version requires again the use of an additional bitmap together with the support of the BP data structure. Recall that, when working with `parent` axis, traversed occurrences of R marked their parents as a way to keep them *memorized* in order to qualify further occurrences of L. We also apply the same strategy for $\texttt{parent}_{\texttt{dist}}$. This time, however, we are not interested in the parent of a right segment, but in the ancestor at distance d. Hence, here we can use the *level_ancestor* operation provided by our BP representation, instead of the *enclose* one we used there (see lines 14 and 20 of Algorithm 8.14): $pos\_s_{parent} \leftarrow enclose(right.s) \Longrightarrow pos\_s_{parent} \leftarrow level\_ancestor(right.s, d)$.

Ancestor$_{\mathbf{dist}}$

In this case not only ancestors at distance d from a target element are valid, but also those at a greater distance. Therefore, similar schemas to that used for $\texttt{parent}_{\texttt{dist}}$ can also be applied for this operator, with minor changes. For instance, if we are in *non-nested* scenario, the variation comes from the use of a different comparison inequality during the check of the depth level condition: $depth(right.s){=}(depth(left.s) + d) \Longrightarrow depth(right.s){\geq}(depth(left.s) + d)$. On the other hand, and regarding *full-nested* variant, now each traversed occurrence

[14]Note that L_2 will be compared with R_2, which is the current segment of the right side, at this moment.

of R must mark the ancestor at distance d, along with the rest of its ancestors up to the root. That is, right segments must flag ancestors at distance d or more. Thus, the use of $level\_ancestor$ is extended to cover all of them: $level\_ancestor(right.s, i)\ \forall i\ = d \ldots depth(right.s)$.

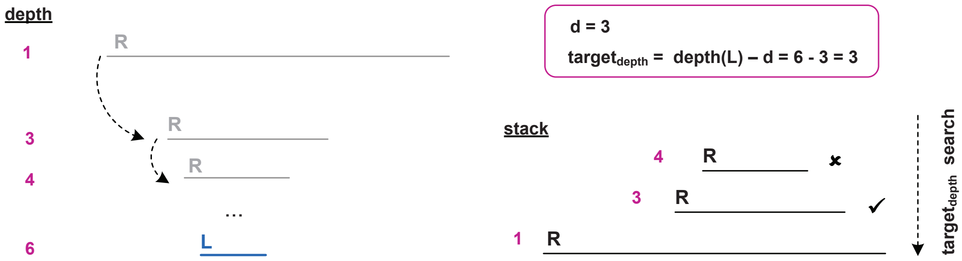


Figure 8.7: Example for the *full-nested* variant of `child`$_{\texttt{dist}}$ axis.

Algorithm 8.17: Modification to be applied over *full-nested* variant of `child` operator to meet `child`$_{\texttt{dist}}$ semantics

while $left \not\subseteq top(stack)$ **do** // stack update
  $pop(stack)$
if ! $isEmpty(stack)$ **then** // $target_{depth}$ search
  $i \leftarrow 0$; $depth_{stack} \leftarrow getDepth(stack, i)$
  while $depth_{stack} > target_{depth}$ **and** $i < size(stack)$ **do**
    $i \leftarrow i + 1$; $depth_{stack} \leftarrow getDepth(stack, i)$
else
  $depth_{stack} \leftarrow -1$
if $depth_{stack} = target_{depth}$ **then**
  $left_s \leftarrow left.s + 1$; $result \leftarrow left$; **return** $result$
else
  case ...

Child$_{\textbf{dist}}$

`Child`$_{\texttt{dist}}$ looks for elements at distance d descending from a target segment. Thus, similarly to `parent`$_{\texttt{dist}}$ with regards to `parent`, *non-nested* variant of `child`$_{\texttt{dist}}$ emulates the corresponding *non-nested* version of `child`, but including the d distance parameter to validate an occurrence of L when `left`$\subset$`right` (see line 11 of Algorithm 8.15): $depth(left.s) = (depth(right.s) + 1) \implies depth(left.s) = (depth(right.s) + d)$. Likewise, *full-nested* alternative follows an equivalent approach to `child` axis for the same scenario, since a stack is also needed to store the ancestors of *right* that have already been traversed. Yet, to deliver an occurrence of L the condition to be fulfilled is slightly modified. Note that *left* could be selected whenever there is an occurrence of R in the stack whose depth level matches $target_{depth}$ =*depth(left.s)* $- d$ (see Figure 8.7). Hence, once the stack is updated, we must look for that occurrence. Algorithm 8.17 presents the fragment of pseudocode by which lines from 6 to 10 and lines from 24 to 28 should be replaced in the just seen Algorithm 8.16, to consider that new feature. Notice that we do not need to inspect all the segments of the stack, since depth levels descend as we deepen into the stack (by definition). Therefore, we stop searching when we reach a depth level equal to $target_{depth}$, or even lower than it.

Descendant$_{\texttt{dist}}$

`Descendant`$_{\texttt{dist}}$ is to `child`$_{\texttt{dist}}$ as `ancestor`$_{\texttt{dist}}$ is to `parent`$_{\texttt{dist}}$. That is, *left* comes a valid result if it has an ancestor of type R at distance d or more. For instance, let us consider the same example illustrated in Figure 8.7, but now assuming that $d = 4$. If we use `child`$_{\texttt{dist}}$ operator, the occurrence of L depicted in blue does not qualify under this condition, as there is no occurrence of R at distance $target_{depth} = 6 - 4 = 2$. However, the same occurrence of L will be sent upwards in case of `descendant`$_{\texttt{dist}}$. Note that, in this situation, it is enough for L segment, an occurrence of R being at least 4 levels above it. Thus, the occurrence whose depth is 1 makes L qualify. This difference results into simple changes over the depth check conditions, for both variants, regarding `child`$_{\texttt{dist}}$ pseudocodes. In case of *non-nested* scenario the equality comparison turns into $\geq$: $depth(left.s){=}(depth(right.s) + d) \Longrightarrow depth(left.s){\geq}(depth(right.s) + d)$. On the other hand, *full-nested* variant replaces $depth_{stack}{=}target_{depth}$ (see line 9 of Algorithm 8.17) by $depth_{stack}{\leq}target_{depth}$.

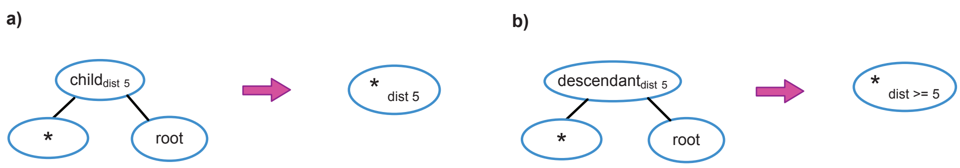


Figure 8.8: Special cases of use of `child`$_{\texttt{dist}}$ and `descendant`$_{\texttt{dist}}$.

Child$_{\texttt{dist}}$ and Descendant$_{\texttt{dist}}$ optimizations

As leaf operators, for which we noted special queries where general *next* procedures could be optimized, some of the previously discussed parameterized operators can also be performed more efficiently. In particular, `child`$_{\texttt{dist}}$ and `descendant`$_{\texttt{dist}}$ operators, as long as they are used in queries like `/*/*/*/*/*` (see Figure 8.8 a)) and `/*/*//*/*` (see Figure 8.8 b)), respectively. Note that in these situations, both operators eventually lead to leaf nodes that must cover the occurrences of elements with given depths. Therefore, we could take advantage of some of the tree operations provided by the balanced parentheses data structure: *i*) *level_leftmost*, which obtains the leftmost node (an element for us, since the tree is defined by the XML document structure) with a given depth; *ii*) *level_next*, that gets the next node (element) of another one in BFS[15] order.

Algorithm 8.18: *Next* procedure of *any* element of depth d

Input: d (depth), $last_e$ (end position of the last delivered result)
Output: next occurrence of an element of depth d
// 1^{st} call
$pos_s \leftarrow level\_leftmost(d)$
$pos_e \leftarrow findclose(pos_s)$
// Next calls
$pos_s \leftarrow level\_next(last_e)$
$pos_e \leftarrow findclose(pos_s)$
$result \leftarrow segment(pos_s, pos_e)$; **return** $result$

Algorithm 8.19: *Next* procedure of *any* element of depth $\geq d$

Input: d (depth), $last_e$ (end position of the last delivered segment of depth d)
Output: next occurrence of an element of depth $\geq d$
// 1^{st} call
$pos_s \leftarrow level\_leftmost(d)$
$pos_e \leftarrow findclose(pos_s)$
$find\_descendants(queue, pos_s)$
$result \leftarrow segment(pos_s, pos_e)$
// Next calls
if ! $isEmpty(queue)$ **then**
    $result \leftarrow segment(pos_,pos_e)$
else
    $pos_s \leftarrow level\_next(last_e)$
    $pos_e \leftarrow findclose(pos_s)$
    $find\_descendants(queue, pos_s)$
    $result \leftarrow segment(pos_s, pos_e)$
return $result$

In case of the operator deduced from `child`$_{\texttt{dist}}$ semantics, *next* procedure just receives the end position of the last delivered result, besides d distance. Then, the

[15]BFS: breadth first search.

first call locates the position of the first element in the XML document whose depth is d. Next calls to the same function deliver subsequent occurrences of elements also fulfilling the same depth condition. Algorithm 8.18 shows the pseudocode. If we are in the second scenario, elements of depth d, but also greater than d, are valid. Thus, each time an occurrence of depth d is found, its descendants are stored into a queue, from which segments are then delivered until it becomes empty. At this moment, the next occurrence of depth d is located, and the whole procedure is repeated. The pseudocode is presented in Algorithms 8.19 and 8.20.

Algorithm 8.20: $find\_descendants$ procedure

Input: $queue$ (queue of descendants), $target_s$ (start position of the element whose descendants must be located)
Output: $queue$ filled with the descendants of $target_s$

$child_s \leftarrow first\_child(target_s)$
while $child_s \neq -1$ **do**
    $push(queue, child_s, findclose(child_s))$
    $find\_descendants(queue, child_s)$
    $child_s \leftarrow next\_sibling(child_s)$

Algorithm 8.21: *Next* procedure of `following` operator (*non-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `following` semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
    case $left > right$
        $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
    otherwise
        $left_s \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return $result$

8.2.2.6 Following

`Following` axis constitutes an special operator that always advances to the next left segment, once fixed the *correct* right one. In case of *non-nested* elements, this *correct* right segment matches the first occurrence of R. Yet, for *full-nested* variant it may yield a problem.

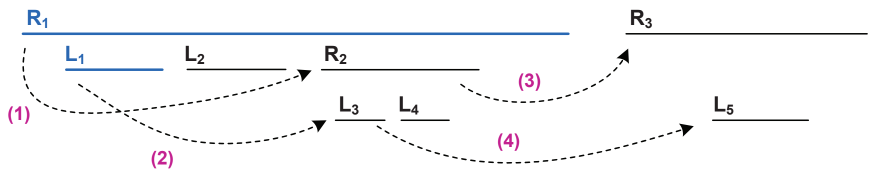


Figure 8.9: Example for `following` axis.

Algorithm 8.22: *Next* procedure of `following` operator (*full-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `following` semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
// Initialization: $right_{prev}$ is fixed to the *correct* R segment
// $right_s \leftarrow 1$; $right_{prev} \leftarrow R.next(right_s, right_e)$
// $right_s \leftarrow right_{prev}.s + 1$; $right_{next} \leftarrow R.next(right_s, right_e)$
// **while** $right_{next}.e < right_{prev}.e$ **do**
// $right_{prev} \leftarrow right_{next}$
// $right_s \leftarrow right_{next}.s + 1$; $right_{next} \leftarrow R.next(right_s, right_e)$
// **end**
$right \leftarrow right_{prev}$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left > right$
    $left_s \leftarrow left.s + 1$; $result \leftarrow left$; **return** $result$
  otherwise
    $left_s \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return $result$

Let us consider the example of Figure 8.9. The first occurrence of R is R_1, which is precisely the current right segment, together with L_1, the current occurrence of the left side. Hence, given their relationship $L_1 \subset R_1$, we should move to the next occurrence of R after R_1 start position, since a nested occurrence or R may still occur before L_1. This leads to R_2 segment. Now, $L_1 < R_2$, so we advance to the next left segment starting after R_2 beginning, that is, L_3. Again, we are in a situation similar to the initial one, thus we proceed in the same way, and move to R_3, which also causes L to be advanced to L_5 segment (striped arrows in Figure 8.9 indicate the flow of movements). Notice that L_5 already fulfills `following` semantics, as it appears after R_1 and even R_2. However, given current segments, R_3 and L_5, we can not detect L_5 as a valid result: $L_5 \subset R_3$, thus a right advance would be applied.

Therefore, the solution is to fix the *correct* right segment at the beginning, from which then we can start a general *next* procedure. This target right segment is the furthest occurrence of R, not starting after the end of any other right segment. In the example of Figure 8.9 this occurrence is represented by R_2. The complete pseudocode describing both variants is presented in Algorithms 8.21 and 8.22.

Algorithm 8.23: *Next* procedure of `preceding` operator (*non-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `preceding` semantics

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
  case $left \subseteq right$
    $right_s \leftarrow right.e + 1$; $right \leftarrow R.next(right_s, right_e)$
  otherwise
    $right_s \leftarrow left.e + 1$; $right \leftarrow R.next(right_s, right_e)$
$result \leftarrow \emptyset$
return $result$

Figure 8.10: Example for `preceding` axis.

8.2.2.7 Preceding

If we deal with *non-nested* version, `preceding` performance is based on the advance of segments from the right side whenever current segments do not satisfy `left<right`. At this moment, we move to the next left segment, until it overcomes/intersects *right*. Algorithm 8.23 describes the pseudocode.

In turn, *full-nested* variant is not as simple, since we must be aware of whether *right* holds the last occurrence of R. As shown in Figure 8.10, given current left and right segments, L_1 and R_1, respectively, if R_1 is not the last occurrence, we could advance to the next right segment. Note that any other forward occurrence of R would qualify the same left segments than R_1, and maybe some additional ones.

For instance, if we assume that there is a segment such as R_2, it would make L_2 and L_3 be selected (as R_1), but also L_1. Nevertheless, if R_1 is the last one, instead of moving to the next right segment, we should keep it and advance the left side, since L_2 and L_3 can still be delivered. The pseudocode for this scenario is presented in Algorithm 8.24.

Algorithm 8.24: *Next* procedure of `preceding` operator (*full-nested* variant)

```
Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling preceding semantics
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
right ← R.result
while left ≠ø and right ≠ø do
    case left < right
        left_s ← left.s + 1; result ← left; return result
    case left > right
        right_s ← left.s + 1; right ← R.next(right_s, right_e)
    case left ⊆ right
        if ! rLastFound then
            right_s ← left.s + 1; right ← R.next(right_s, right_e)
        else
            break;
    otherwise
        if ! rLastFound then
            right_last ← right; right_s ← right.s + 1; right ← R.next(right_s, right_e)
            if right =ø then
                right ← right_last; rLastFound ← 1
                left_s ← left.s + 1; left ← L.next(left_s, left_e)
        else
            left_s ← left.s + 1; left ← L.next(left_s, left_e)
result ← ø
return result
```

A different approach, with respect to the general performance of internal nodes, can be assumed by `preceding` axis, when the right child is a leaf node. In that case, regardless of working with elements that are self-nested or not, `preceding` operator is solved by directly locating the last occurrence of the right side, and then using it as a limit up to which left segments can advance. The pseudocodes of *non-nested* and *full-nested* variants under these conditions are shown in Algorithms 8.25 and 8.26, respectively.

Algorithm 8.25: Special *next* procedure of `preceding` operator (*non-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `preceding` semantics

```
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
// Initialization: right_last is set to the last occurrence of R
// occ_s_right ← count(tag_s_right, XWTroot_length)
// pos_s_right ← locate(tag_s_right, occ_s_right)
// pos_e_right ← findclose(pos_s_right)
// right_last ← segment(pos_s_right, pos_e_right)
right ← right_last
while left ≠ ø do
    case left < right
        left_s ← left.e + 1; result ← left; return result
    otherwise
        break;
result ← ø
return result
```

Algorithm 8.26: Special *next* procedure of `preceding` operator (*full-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `preceding` semantics

```
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
// Initialization: right_last is set to the last occurrence of R
// occ_s_right ← count(tag_s_right, XWTroot_length)
// pos_s_right ← locate(tag_s_right, occ_s_right)
// pos_e_right ← findclose(pos_s_right)
// right_last ← segment(pos_s_right, pos_e_right)
right ← right_last
while left ≠ ø do
    case left < right
        left_s ← left.s + 1; result ← left; return result
    case left ⊃ right
        left_s ← left.s + 1; left ← L.next(left_s, left_e)
    otherwise
        break;
result ← ø
return result
```

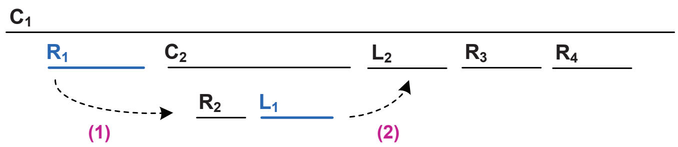


Figure 8.11: Example for `following-sibling` axis.

8.2.2.8 Following-sibling

Similarly to *full-nested* variant of `parent` axis, where the parents of traversed R segments should be *remembered* through a bitmap to qualify forward occurrences of L, `following-sibling` also makes use of an auxiliary structure. This time, the bitmap is replaced by a hash table. Notice that now, unlike `parent`, to know that a given segment has been marked by a child of type R is not enough. Actually, the parent element should be stored together with the right segment that caused it to be flagged. Indeed, if an element has more than one children of type R, only the leftmost one should be recorded, according to `following-sibling` semantics. Hence, whenever an occurrence of R is traversed, we search for its parent in the hash table. If it is not found, we store the parent together with the own right segment. Otherwise, we do not perform any additional process (since it means that a previous right segment has already marked the same parent). Figure 8.11 illustrates an example. Being L_1 and R_1 current left and right segments, respectively, we need to advance to the next right segment starting after the end of the current one. Thus, we reach R_2 and update the hash table with the information about its parent, that is, C_2. Since L_1 parent matches C_2 and L_1 is after R_2, L_1 becomes a valid result, which is sent upwards. The next call moves to the next occurrence of L_1, in that example, L_2, and then looks for its parent, C_1, in the hash table. Recall that this segment was kept along with R_1, when this last one was traversed. This allows L_2 to be now qualified.

The explained solution applies regardless the variant of `following -sibling`. The pseudocode of *full-nested* procedure is described in Algorithm 8.21. Note as well that once the last occurrence of R has been visited, left segments can still qualify, as long as they precede the end position of the furthest parent classified by a right segment.

8.2.2.9 Preceding-sibling

The same approach discussed for `following-sibling` can also be applied for `preceding-sibling` axis. Yet, in this case, the hash table does not store the earlier R child for a given parent, but the last one encountered at each moment.

Algorithm 8.27: *Next* procedure of `following-sibling` operator (*full-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `following-sibling` semantics

```
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
right ← R.result
while left ≠ ø and right ≠ ø do
    if hash[search(enclose(left.s))].pos_s_child < left.s then
        left_s ← left.s + 1; result ← left; return result
    else
        case left < right
            left_s ← left.e + 1; left ← L.next(left_s, left_e)
        case left > right
            pos_e_rparent ← findclose(enclose(right.s))
            if left.s > pos_e_rparent then
                right_s ← pos_e_rparent + 1; right ← R.next(right_s, right_e)
            else
                right_s ← right.e + 1; right ← R.next(right_s, right_e)
            if search(enclose(right.s)) = −1 then
                max_e_rparent ← max(max_e_rparent, pos_e_rparent)
                insert(enclose(right.s), right.s)
        case left ⊂ right
            right_s ← right.s + 1; right ← R.next(right_s, right_e)
            if search(enclose(right.s)) = −1 then
                pos_e_rparent ← findclose(enclose(right.s))
                max_e_rparent ← max(max_e_rparent, pos_e_rparent)
                insert(enclose(right.s), right.s)
        otherwise
            left_s ← right.s + 1; left ← L.next(left_s, left_e)
while left ≠ ø and left.s < max_e_rparent do
    if hash[search(enclose(left.s))].pos_s_child < left.s then
        left_s ← left.s + 1; result ← left; return result
    else
        left_s ← left.s + 1; left ← L.next(left_s, left_e)
result ← ø
return result
```

Thus, whenever we advance to a new right segment, although its parent had already been marked by a previous occurrence of R, we update it with the information of the just one found. The rest of the actions performed under the different segments comparison scenarios work in line with `preceding-sibling` semantics. Algorithm 8.28 shows the pseudocode for the *full-nested* variant.

Algorithm 8.28: *Next* procedure of `preceding-sibling` operator (*full-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `preceding-sibling` semantics

```
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
right ← R.result
while left ≠ ø and right ≠ ø do
    if hash[search(enclose(left.s))].pos_s_child > left.s then
        left_s ← left.s + 1; result ← left; return result
    else
        case left < right
            if right.s > findclose(enclose(left.s)) then
                left_s ← left.s + 1; left ← L.next(left_s, left_e)
            else
                max_s_right ← right.s
                right_s ← right.s + 1; right ← R.next(right_s, right_e)
                insert(enclose(right.s), right.s)
        case left ⊃ right
            max_s_right ← right.s
            right_s ← right.s + 1; right ← R.next(right_s, right_e)
            insert(enclose(right.s), right.s)
        otherwise
            max_s_right ← right.s
            right_s ← left.s + 1; right ← R.next(right_s, right_e)
            insert(enclose(right.s), right.s)
while left ≠ ø and left.s < max_s_right do
    if hash[search(enclose(left.s))].pos_s_child > left.s then
        left_s ← left.s + 1; result ← left; return result
    else
        left_s ← left.s + 1; left ← L.next(left_s, left_e)
result ← ø
return result
```

8.2.2.10 Basic Operators over Attributes

Up to now, the above mentioned internal nodes worked over child nodes delivering elements. But some of the axes also apply for attributes (e.g. $\texttt{ancestor}_{\texttt{att}}$,

$\texttt{descendant}_{\texttt{att}}$, $\texttt{parent}_{\texttt{att}}$, and $\texttt{child}_{\texttt{att}}$). In those cases, special procedures are devised to deal with them.

Notice that unlike elements, for which the segment representation arose from positions in the *XDTree* node, attribute representation regards positions in the text, that is, in the root of the XWT. As a result, we have to perform some conversions to make both work together. For instance, segment comparisons will be made, as in general *next* procedures of internal nodes, regarding positions in the *XDTree* node. Thus, attribute positions in the root node must be converted to positions in the *XDTree*, which turn into a representation that matches the start-tag position of the element holding that attribute. Let us assume that b_y is the byte reserved to be the first byte of start/end-tags, and that we are working with an occurrence of an attribute name placed at position p in the root node. Then, $p_{XDTree} = rank_{b_y}(XWTroot, p)$ give us its segment representation in the *XDTree* node, $[p_{XDTree}, p_{XDTree}]$. Also observe that, at this level, the attribute representation always stands for a point, since its segment will start and finish at the same position. Indeed, the typical five different segment relations come to four, as elements containment makes no sense.

Moreover, also segments advance may impose positional conversions in some cases. Note that the request of new segments must be made in accordance with the representation used for each kind of component. Hence, whenever attributes *skipping* is determined by element positions, we must perform their transformation to gather the actual text positions[16], from which the attributes advance is then performed. The same happens in the reverse scenario, but regarding attribute positions in the *XDTree* node, to perform elements *skipping*.

$\textbf{Ancestor}_{\texttt{att}}$

This operator delivers elements (the left side) that either have the target attribute (the right side) or hold any descendant having it. Therefore, similarly to `ancestor`, whenever `left<right`, we advance to the next left segment whose end position finishes after the *right* start. In turn, if `left>right`, we move to the next attribute segment starting after *left* beginning. Finally, if `left⊇right`, then *left* becomes a valid result. The update of the new positional restrictions is the same for both variants (that is, for the *non-nested* and *full-nested* variants), under the first two comparison scenarios. Yet they differ regarding the last situation (that is, if `left⊇right`). In case of *full-nested* version, once a valid left segment is found, it still may contain some other nested occurrences also fulfilling the $\texttt{ancestor}_{\texttt{att}}$ semantics. Thus the update is made accordingly, that is, as performed for `left<right`. Algorithms 8.29 and 8.30 show the pseudocode of both variants.

[16]Notice that we only need to perform an additional *select* operation from the current start (s)/end (e) position in the XDTree node, to find the corresponding one in the root of the XWT: $s_{root} = select_{b_y}(XWTroot, s)$ / $e_{root} = select_{b_y}(XWTroot, e)$.

Algorithm 8.29: *Next* procedure of `ancestor`$_{\texttt{att}}$ operator (*non-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `ancestor`$_{\texttt{att}}$ semantics
// We assume that b_y is the first byte of a start/end-tag codeword

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_e \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
    $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
    if $right \neq \emptyset$ **then**
      $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$
  otherwise
    $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
$result \leftarrow \emptyset$
return $result$

Algorithm 8.30: *Next* procedure of `ancestor`$_{\texttt{att}}$ operator (*full-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `ancestor`$_{\texttt{att}}$ semantics
// We assume that b_y is the first byte of a start/end-tag codeword

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_e \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
    $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
    if $right \neq \emptyset$ **then**
      $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$
  otherwise
    $left_e \leftarrow right.s + 1$; $result \leftarrow left$; **return** $result$
$result \leftarrow \emptyset$
return $result$

Algorithm 8.31: *Next* procedure of $\texttt{descendant}_{\texttt{att}}$ operator (applicable for *non-nested* and *full-nested* variants)

Input: new_s (new positional restriction)
Output: next occurrence of the left side fulfilling $\texttt{descendant}_{\texttt{att}}$ semantics
// We assume that b_y is the first byte of a start/end-tag codeword

$left_s \leftarrow max(left_s, new_s)$
$left \leftarrow L.next(left_s)$
if $left.s_{root} \neq \emptyset$ **then**
  $left.s \leftarrow rank_{b_y}(XWTroot, left.s_{root})$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $right.s_{root} \leftarrow select_{b_y}(XWTroot, right.s)$
    $left_s \leftarrow right.s_{root} + 1$; $left \leftarrow L.next(left_s)$
    if $left \neq \emptyset$ **then** $left.s \leftarrow rank_{b_y}(XWTroot, left.s_{root})$
  case $left > right$
    $right_e \leftarrow left.s + 1$; $right \leftarrow R.next(right_s, right_e)$
  otherwise
    $left_s \leftarrow left.s_{root} + 1$; $result \leftarrow left$; **return** $result$
$result \leftarrow \emptyset$; **return** $result$

$\text{Descendant}_{\text{att}}$

$\texttt{Descendant}_{\texttt{att}}$ reverses $\texttt{ancestor}_{\texttt{att}}$ semantics. Hence, in that case, we select an attribute if it corresponds to the target element or to any of its descendants. Note that now L denotes attribute occurrences, while R is representing element segments. The *next* procedure presented in Algorithm 8.31 is used to perform the *non-nested* variant, but also the *full-nested* one, unlike `descendant` axis, since there differences between both variants mainly resulted from the fact that the left side could be self-nested. This situation does not apply for attributes.

$\text{Parent}_{\text{att}}$

Unlike `parent`, for which we need to remember the parents of traversed nested occurrences of R by using a bitmap, here that problem does not crop up, as the right node stands for attribute occurrences. Therefore, the performance of $\texttt{parent}_{\texttt{att}}$ is quite similar to $\texttt{ancestor}_{\texttt{att}}$, but with minor changes, if we consider that now we are not looking for left segments containing the current attribute, but just the occurrence of L whose start position precisely matches the position of the attribute in the *XDTree* node. That is, the exact element that holds the current attribute. As a result, if `left<right`, both *non-nested* and *full-nested* variants advance to the next left segment beginning at *right* start, converted to a position in the *XDTree* node[17], while in the opposite scenario (i.e. `left>right`) it is the right one which

[17] Recall that the segment representation of elements refers to positions in this branch.

Algorithm 8.32: *Next* procedure of $\texttt{parent}_{\texttt{att}}$ operator (*non-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling $\texttt{parent}_{\texttt{att}}$ semantics
// We assume that b_y is the first byte of a start/end-tag codeword

```
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
right ← R.result
while left ≠ ∅ and right ≠ ∅ do
    case left < right
        left_s ← right.s; left ← L.next(left_s, left_e)
    case left > right
        left.s_root ← select_{b_y}(XWTroot, left.s)
        right_s ← left.s_root + 1; right ← R.next(right_s)
        if right ≠ ∅ then right.s ← rank_{b_y}(XWTroot, right.s_root)
    case left ⊃ right
        left_s ← left.e + 1; left ← L.next(left_s, left_e)
    otherwise
        left_s ← left.e + 1; result ← left; return result
result ← ∅
return result
```

Algorithm 8.33: *Next* procedure of $\texttt{parent}_{\texttt{att}}$ operator (*full-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling $\texttt{parent}_{\texttt{att}}$ semantics
// We assume that b_y is the first byte of a start/end-tag codeword

```
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
right ← R.result
while left ≠ ∅ and right ≠ ∅ do
    case left > right
        left.s_root ← select_{b_y}(XWTroot, left.s)
        right_s ← left.s_root + 1; right ← R.next(right_s)
        if right ≠ ∅ then right.s ← rank_{b_y}(XWTroot, right.s_root)
    case left.s = right.s
        left_s ← left.s + 1; result ← left; return result
    otherwise
        left_s ← right.s; left ← L.next(left_s, left_e)
result ← ∅
return result
```

is moved to that appearing after *left* start. Note that this time *left* start must be converted to a position in the XWT root node, according to attributes representation. In turn, for `left⊃ right`, *full-nested* variant makes the left side obtain the next occurrence that may occur inside *left*, holding the current attribute, whereas in *non-nested* version, we are sure that this occurrence does not exist, hence we simply advance to the next left segment starting after the end of *left*. The remaining comparison case, `right.s=left.s`, delivers *left* upwards in both scenarios, also updating the start positional restriction of new retrieved left segments to the current *left* start (in case of *full-nested* variant) or end (in case of the *non-nested* variant), as applicable. We can see the pseudocodes of *non-nested* and *full-nested* versions in Algorithm 8.32 and Algorithm 8.33, respectively.

Algorithm 8.34: *Next* procedure of $\texttt{child}_{\texttt{att}}$ operator (*non-nested* variant)

Input: $news_s$ (new positional restriction)
Output: next occurrence of the left side fulfilling $\texttt{child}_{\texttt{att}}$ semantics
// We assume that b_y is the first byte of a start/end-tag codeword

$left_s \leftarrow max(left_s, news_s)$
$left \leftarrow L.next(left_s)$
if $left.s_{root} \neq \emptyset$ **then**
  $left.s \leftarrow rank_{b_y}(XWTroot, left.s_{root})$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $right.s_{root} \leftarrow select_{b_y}(XWTroot, right.s)$
    $left_s \leftarrow right.s_{root} + 1$; $left \leftarrow L.next(left_s)$
    if $left \neq \emptyset$ **then**
      $left.s \leftarrow rank_{b_y}(XWTroot, left.s_{root})$
  case $left > right$
    $right_s \leftarrow left.s$; $right \leftarrow R.next(right_s, right_e)$
  case $left \subset right$
    $right_s \leftarrow right.e + 1$; $right \leftarrow R.next(right_s, right_e)$
  otherwise
    $left_s \leftarrow left.s_{root} + 1$; $result \leftarrow left$; **return** $result$
$result \leftarrow \emptyset$
return $result$

$\textbf{Child}_{\textbf{att}}$

Again, the fact of attributes not being *self-nested* makes the use of a stack unnecessary, unlike what happened to `child` axis. On the other hand, and likewise $\texttt{parent}_{\texttt{att}}$ with regard to $\texttt{ancestor}_{\texttt{att}}$, $\texttt{child}_{\texttt{att}}$, also performs similarly to `descendant`$_{\texttt{att}}$. Yet, we still distinguish different *next* procedures depending on whether we are working with self-nested elements or not. Anyway, the main differences regarding `descendant`$_{\texttt{att}}$ stem from the own `child`$_{\texttt{att}}$ semantics, which searches for occurrences of attributes that qualify the target element, and do not any of its descendants. Thus, whenever `left>right`, we do not advance to the next right segment containing the current attribute, but to the next occurrence from the right side that exactly matches the attribute start position in the *XDTree* node, if it exists. In a same way, if segments are related through a descendant relationship (i.e. `left`$\subset$ `right`), *full-nested* variant still tries to find an exact match within the nested occurrences that may occur inside *right*. In case of *non-nested* version, this does not apply and we simply move to the next right segment after the current one. Algorithms 8.34 and 8.35 show the pseudocodes of both variants for `child`$_{\texttt{att}}$ operator.

Algorithm 8.35: *Next* procedure of `child`$_{\texttt{att}}$ operator (*full-nested* variant)

Input: new_s (new positional restriction)
Output: next occurrence of the left side fulfilling `child`$_{\texttt{att}}$ semantics
// We assume that b_y is the first byte of a start/end-tag codeword

$left_s \leftarrow max(left_s, new_s)$
$left \leftarrow L.next(left_s)$
if $left.s_{root} \neq \emptyset$ **then**
    $left.s \leftarrow rank_{b_y}(XWTroot, left.s_{root})$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
    case $left < right$
        $right.s_{root} \leftarrow select_{b_y}(XWTroot, right.s)$
        $left_s \leftarrow right.s_{root} + 1$; $left \leftarrow L.next(left_s)$
        if $left \neq \emptyset$ **then**
            $left.s \leftarrow rank_{b_y}(XWTroot, left.s_{root})$
    case $left.s = right.s$
        $left_s \leftarrow left.s_{root} + 1$; $result \leftarrow left$; **return** $result$
    otherwise
        $right_s \leftarrow left.s$; $right \leftarrow R.next(right_s, right_e)$
$result \leftarrow \emptyset$
return $result$

8.2.2.11 Parameterized Operators over Attributes: the *distance* parameter

The operators previously discussed in Section 8.2.2.5, for which a *distance* parameter was used, can also be extended to work with attributes. Hence we distinguish as well $\texttt{parent}_{\texttt{att\_dist}}$, $\texttt{ancestor}_{\texttt{att\_dist}}$, $\texttt{child}_{\texttt{att\_dist}}$, and $\texttt{descendant}_{\texttt{att\_dist}}$. For each operator, both *non-nested* and *full-nested* variants follow the same guidelines and remarks made for their respective counterparts in Section 8.2.2.5[18]. That is, *non-nested* versions introduce simple depth validations, while *full-nested* ones work with additional auxiliary structures (e.g. a bitmap in case of $\texttt{parent}_{\texttt{att\_dist}}$, and $\texttt{ancestor}_{\texttt{att\_dist}}$, and a stack, in case of $\texttt{child}_{\texttt{att\_dist}}$, and $\texttt{descendant}_{\texttt{att\_dist}}$). Thus, we do not explain them again. We refer the reader to Section 8.2.2.5, for a new review if needed.

Algorithm 8.36: *Next* procedure of `and` (`self`) operator (*non-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `and` semantics
// Note that `left⊂right` and `left⊃right` make no sense for this variant

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_s \leftarrow right.s$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $right_s \leftarrow left.s$; $right \leftarrow R.next(right_s, right_e)$
  otherwise // left=right
    $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
$result \leftarrow \emptyset$
return $result$

8.2.2.12 And

`And` operator searches for same segments. As happened with some of the previous operators, we devise different *next* procedures depending on whether it is applied over elements or over attributes. We denote them as `and` and $\texttt{and}_{\texttt{att}}$, respectively, according to the notation used until now.

In case of elements, `and` also stands for `self` axis. Algorithms 8.36 and 8.37 present the pseudocodes of *non-nested* and *full-nested* variants. Notice that, in both scenarios, the procedure always requests a new segment from the side whose current segment appears before, using as restriction the start position of the segment from the other side, as a way to meet the equality relationship.

[18] Although now also combined together with the attribute features described at the beginning of Section 8.2.2.10.

Algorithm 8.37: *Next* procedure of `and` (`self`) operator (*full-nested* variant)

Input: new_s, new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `and` semantics
// In this scenario, `left⊂right` and `left⊃right` must be considered, since child
// nodes may deliver occurrences of elements regardless its type (i.e. when
// any of them works with '`*`')

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$ **or** $left \supset right$
    $left_s \leftarrow right.s$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$ **or** $left \subset right$
    $right_s \leftarrow left.s$; $right \leftarrow R.next(right_s, right_e)$
  otherwise // left=right
    $left_s \leftarrow left.s + 1$; $result \leftarrow left$; **return** $result$
$result \leftarrow \emptyset$
return $result$

On the other hand, and related to $\texttt{and}_{\texttt{att}}$, the same strategy is used. Algorithm 8.38 describes the pseudocode in that case. Observe that unlike some of the operators previously discussed in Section 8.2.2.10 and Section 8.2.2.11, no positional conversions are needed, since $\texttt{and}_{\texttt{att}}$ does not combine the use of elements with attributes, and hence all positions are referred to locations in the root of the XWT.

Algorithm 8.38: *Next* procedure of $\texttt{and}_{\texttt{att}}$ operator

Input: new_s (new positional restriction)
Output: next occurrence of the left side fulfilling $\texttt{and}_{\texttt{att}}$ semantics

$left_s \leftarrow max(left_s, new_s)$; $left \leftarrow L.next(left_s)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_s \leftarrow right.s_{root}$; $left \leftarrow L.next(left_s)$
  case $left > right$
    $right_s \leftarrow left.s_{root}$; $right \leftarrow R.next(right_s)$
  otherwise // left=right
    $left_s \leftarrow left.s_{root} + 1$; $result \leftarrow left$; **return** $result$
$result \leftarrow \emptyset$
return $result$

Algorithm 8.39: *Next* procedure of `or` operator (*full-nested* variant)

```
Input: new_s, new_e (new positional restrictions)
Output: next occurrence of any child
left ← L.result
right ← R.result
if lastL or (left ≠ø and (left.s < new_s or left.e < new_e)) then
    left_s ← max(left_s, new_s)
    left_e ← max(left_e, new_e)
    left ← L.next(left_s, left_e)
if lastR or (right ≠ø and (right.s < new_s or right.e < new_e)) then
    right_s ← max(right_s, new_s)
    right_e ← max(right_e, new_e)
    right ← R.next(right_s, right_e)
lastL ← 0; lastR ← 0
if left ≠ø and right ≠ø then
    case left < right or left ⊃ right
        left_s ← left.s + 1; lastL ← 1; result ← left; return result
    case left > right or left ⊂ right
        right_s ← right.s + 1; lastR ← 1; result ← right; return result
else
    if left ≠ø then
        left_s ← left.s + 1; lastL ← 1; result ← left; return result
    else
        if right ≠ø then
            right_s ← right.s + 1; lastR ← 1; result ← right; return result
        else
            result ←ø; return result
```

8.2.2.13 Or

`Or` constitutes a special operator, since against the rest of the internal nodes, which deliver segments that come from the left side, `or` may deliver occurrences received from any side. Therefore, the segment request performed at the beginning of the *next* procedure will not obtain a new segment from the left side, as usual, but from the last delivered side. What is more, even the side that has not been delivered in the previous call, may also be requested for a new segment together with the corresponding one, in case that the incoming restrictions imply its update. Then, whenever we do not reach the last occurrence from any side, current segments are compared to deliver the one starting first. Any other way, results are directly requested to the unique side from which segments remain to be obtained.

This procedure applies whether we work with elements, or with attributes[19]. Furthermore, as a result of the modifications performed over the *query parse tree* (see *Or/and optimizations* in Section 6.3), `or` operator may also deal with nodes delivering words or even phrases[20]. Again, the same general guidelines are followed in those cases. We next show the pseudocode of *full-nested* variant of the `or` operator over elements (see Algorithm 8.39). The rest of the pseudocodes are presented in Appendix B (see Algorithm B.2, Algorithm B.3 and Algorithm B.4).

Algorithm 8.40: *Next* procedure of `contains` text function for single words (*full-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `contains` semantics
// We assume that b_y is the first byte of a start/end-tag codeword

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
    case $left < right$
        $left_e \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
    case $left > right$
        $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
        $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
        if $right \neq \emptyset$ **then**
            $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$
    case $left \supseteq right$ // left.s<=right.s **and** left.e>right.e
        $left_e \leftarrow right.s + 1$; $result \leftarrow left$; **return** $result$
    otherwise
        $left_s \leftarrow left.e + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return $result$

8.2.2.14 Text Functions: `contains` and `equal`

Equality and containment functions follow quite different approaches depending on whether they are applied over elements or attributes. Hence we will refer to them separately to discuss the main features of each one.

- Let us start with text functions over elements. Similarly to what happens when we work with operators that combine elements and attributes from

[19] We denote the operator as $\texttt{or}_{\texttt{att}}$ in this case.

[20] In case of words, the same $\texttt{or}_{\texttt{att}}$ procedure can be used. Yet in case of phrases, a new $\texttt{or}_{\texttt{phrase}}$ algorithm is devised.

both children (e.g. `parent`$_{\texttt{att}}$, `descendant`$_{\texttt{att}}$, `child`$_{\texttt{att\_dist}}$, etc.), here the use of words/phrases, makes positional conversions be necessary to perform segment comparisons, as well as to update the positional restrictions used by new segment requests when they are determined by the segment positions of the other side. Like attributes, the conversion of the segment representation of a word to positions in the *XDTree* node leads to a point given, in that case, by the position in that branch of the start/end-tag that immediately precedes it. In turn, phrases may lead to a point, or even to a segment, in case it spans more than one text node (i.e. if there are interleaved start/end-tags)[21].

Algorithm 8.41: *Next* procedure of `contains` text function for a phrase (*full-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `contains` semantics
// We assume that b_y is the first byte of a start/end-tag codeword

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_e \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
    $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
    if $right \neq \emptyset$ **then**
      $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$;
      $right.e \leftarrow rank_{b_y}(XWTroot, right.e_{root})$
  case $left \supseteq right$ // left.s<=right.s **and** left.e>right.e
    $left_e \leftarrow right.e + 1$; $result \leftarrow left$; **return** $result$
  case $left.s > right.s$ **and** $left.e > right.e$
    $right_s \leftarrow right.e_{root} + 1$; $right \leftarrow R.next(right_s)$
    if $right \neq \emptyset$ **then**
      $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$;
      $right.e \leftarrow rank_{b_y}(XWTroot, right.e_{root})$
  otherwise
    $left_s \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return $result$

[21] Notice that if the positional conversion of a phrase representation yields a segment, this does not have to match the limits of a specific element. As a result, more segments comparison scenarios are possible, as overlaps may occur.

Non-nested and *full-nested* variants are possible under this scenario. Algorithm 8.40 and Algorithm 8.41 show the pseudocodes of *full-nested* version for words and phrase containment, respectively[22].

Regarding the `equal` function, the same procedures can be generalized to meet its semantics. This time we need to include an additional check whenever an occurrence of the left side contains the current right segment, to ensure the equality condition, before delivering the left one. This validation may imply the *skipping* of interleaved occurrences of start/end-tags, comments and processing instructions between the boundaries of both segments. Thus, a similar procedure to that explained in Section 8.2.1.3, where these special components were skipped when searching for interleaved phrases, is also applied in that situation (see Algorithms B.7 to B.10 in Appendix B).

Algorithm 8.42: *Next* procedure of $\texttt{contains}_{\texttt{att}}$ text function

Input: $news_s$ (new positional restrictions)
Output: next occurrence of the left side fulfilling $\texttt{contains}_{\texttt{att}}$ semantics
// We assume that b_x is the first byte of an attribute codeword

$left_s \leftarrow max(left_s, news_s)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $att\_to\_left = rank_{b_x}(XWTroot, left.s_{root})$
    $att\_to\_right = rank_{b_x}(XWTroot, right.s_{root})$
    if $(att\_to\_right - att\_to\_left) > 0$ **then**
      $pos\_s_{att\_to\_right} \leftarrow select_{b_x}(XWTroot, att\_to\_right)$
      $left_s \leftarrow pos\_s_{att\_to\_right}$; $left \leftarrow L.next(left_s)$
    else
      $left_s \leftarrow left.s_{root} + 1$; $result \leftarrow left$; **return** $result$
  otherwise
    $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
$result \leftarrow \emptyset$
return $result$

- If we consider the same text functions, but applied over attributes, then $\texttt{contains}_{\texttt{att}}$ and $\texttt{equal}_{\texttt{att}}$ operators arise. Unlike the previous scenario, no positional conversions are performed, since both attribute and word/phrase segments are referred to positions in the same XWT node, that is, to positions in the root of the XWT, in that case.

[22] *Non-nested* versions are described by Algorithm B.5 and Algorithm B.6, in Appendix B.

Algorithm 8.42 shows the pseudocode of $\texttt{contains}_{\texttt{att}}$. Notice that the same pseudocode can be used regardless we are working with words or phrases. In both cases, there are just two different comparison scenarios.

When `left<right`, containment condition is figured out by simply computing the number of attribute occurrences[23] between the start boundaries of each current segment. That is, let us assume that b_x is the byte used to mark the first byte of an attribute codeword, and that s_l and s_r are the start positions of current left and right segments, respectively. Then, $i = rank_{b_x}(XWTroot, s_l)$ gives us the number of attributes before s_l. Likewise, $j = rank_{b_x}(XWTroot, s_r)$, provides the same information but regarding s_r. The containment condition is fulfilled if the subtraction of i from j is equal to 0. If not the case, we request the next occurrence from the left side beginning, at least, at the start position of the j^{th} attribute: $select_{b_x}(XWTroot, att_{s_r})$. Notice that, although we do not know if the j^{th} attribute is an occurrence of the same type as those delivered by the left side, this positional restriction permits to skip all those intermediate left segments that we are sure that are not valid.

In case of any other comparison relationship between current segments (see lines $13 - 14$ in Algorithm 8.42), we advance to the next occurrence of the word/phrase starting after the current attribute segment.

Algorithm 8.43: *Next* procedure of $\texttt{equal}_{\texttt{att}}$ text function

Input: new_s (new positional restrictions)
Output: next occurrence of the left side fulfilling $\texttt{equal}_{\texttt{att}}$ semantics
// We assume that b_x is the first byte of an attribute codeword

$left_s \leftarrow max(left_s, new_s)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
    case $left < right$
        if $right.s_{root} \neq (left.s_{root} + 1)$ **then**
            $left_s \leftarrow right.s_{root} - 1$; $left \leftarrow L.next(left_s)$
        else
            $left_s \leftarrow left.s_{root} + 1$; $result \leftarrow left$; **return** $result$
    otherwise
        $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
$result \leftarrow \emptyset$
return $result$

[23] Regardless they are or not those specified by the left node.

With regards to $\texttt{equal}_{\texttt{att}}$ operator[24] a similar general scheme is used (see Algorithm 8.43). However, this time, in case the attribute starts before the word/phrase (that is, in case `left<right`), just a simple validation must be performed to determine if that attribute becomes a valid result: we only need to check if the word/phrase is placed immediately after the attribute (see line 6 in Algorithm 8.43).

8.2.2.15 Other Functions: `count`

If we are only interested in counting the number of results of a given query, `count` function may be used: `count(`*query*`)`. In a general case, to solve this function, we must first solve the *query* (that is, to locate the different valid results), and then deliver the number of results found. However, this procedure can be optimized for the same set of queries presented in Section 8.2.1.4, such as `//image`, `//*`, `//@author` or even `//@*`. If `count` function is applied over these kind of queries, what we are actually looking for is the number of occurrences of a given element/attribute (likewise, the total number of elements/attributes in the examples that use '`*`'). Therefore, we can save processing time by just performing a count operation of that given pattern (i.e. the specific element or attribute) over the XWT. For instance, let us consider the attribute `author`, whose codeword is $b_x b_i$. Then, `count(//@author)` $= count(b_x b_i, XWTroot_{length})$.

8.2.2.16 Further Discussions

Similarly to what happened with *rank* and *select* operations involved in *count* and *locate* procedures of leaf nodes, the same strategy used there to speed them up, can also be applied to those internal nodes that make use of positional conversions, such as $\texttt{parent}_{\texttt{att}}$, $\texttt{descendant}_{\texttt{att}}$, `equal` or even `contains`. Note that all of them need to perform forward *rank* and *select* operations over the byte reserved to mark the codewords of start/end-tags. Thus, by storing the result for previous operations, we can save processing time.

[24] Remember that one of the first modifications of the *query parse tree* presented in Section 6.3 was the *Attributes equality simplification*, which stands for the transformation of an attribute equality step into an unique text matching operation (in particular, into a *continued phrase* leaf node). Yet, this modification only applies whenever the left side of the equal operator is directly represented by an attribute leaf node, as shown in Figure 6.5. If it corresponds to an internal node, then $\texttt{equal}_{\texttt{att}}$ operator is kept.

Chapter 9

Experimental Evaluation

Chapter 5 to Chapter 7 described in detail our proposal, the XXS system, by focusing on the two main core parts that compose it: the representation module, provided by the XML Wavelet Tree (XWT) data structure, and the query module, for the efficient evaluation of XPath queries over that representation. Now, we present the set of experiments performed to evaluate our work. As a new XML queriable compression tool, both compression properties and querying capabilities have been benchmarked.

Section 9.1 starts by describing the experimental framework used to empirically test the XXS system: the machine used, the collection of documents selected, and the set of queries tested. After that, Section 9.2 focuses on compression features (compression ratio, and compression and decompression times) and presents a large study by comparing our tool with some other general text and XML conscious non-queriable compressors, but also with some well-known solutions supporting XPath, whose space requirements are considered, as well. These last systems are then evaluated again in Section 9.3, this time, regarding their query evaluation performance, to benchmark XXS querying capabilities.

9.1 Experimental Framework

9.1.1 Test Machine

An isolated Intel®Pentium®Core i5 2.67 GHz system, with 16 GB dual-channel DDR-1200Mhz RAM was used in our tests. It ran Ubuntu 11.04 GNU/Linux (kernel version 2.6.38). The compiler used was g++ version 4.5.2 and `-O9` compiler optimizations were set.

9.1.2 Document Corpus

We have collected a large corpus of XML documents selected from multiple data sources. We next present a brief description of the different documents that compose our data set, and point out some of their main properties in Table 9.1. There, the first column indicates the name of the document, while the second and third ones refer to its size (in MBytes), and its maximum depth level, respectively. Then, from column 4 through column 7, the number of different words in the vocabularies of tags (VTags), attributes (VAttributes), text content (VContent), and processing instructions and comments (VNSearch), are shown. Finally, columns 8 to 11, also record the total number of words that hold into each of these vocabularies (see columns tagged as #Tags, #Attributes, #Content, and #NSearch in Table 9.1).

- `XMark`: files generated with *xmlgen*, an XML data generator developed inside *XMark Project*[1]. This tool produces XML documents modelling an auction website, using a parameter $(-f)$ to indicate the size of the documents generated. For our experiments, we created four XML documents using increasing scaling factors.

- `Dblp`: files providing bibliographic information about the most important computer science conferences and publications[2]. The documents used correspond to the revisions of April 2008, and January 2012.

- `Psd`: file belonging to the public proteins database, *Integrated Protein Informatics Resource for Genomic and Proteomic Research*[3]. It contains an integrated collection of proteins functionally annotated.

- `Medline`: files containing bibliographic information about biomedical and life sciences publications[4]. We selected three files of different sizes.

- `Alfred`: file of gene frequency data on human populations supported by the U. S. National Science Foundation[5].

- `Baseball`: document that provides a complete description of baseball statistics of the team players participating in the 1998 Major League.

- `Lineitem`: file providing information about the transactional relational database benchmark *TPC-H* [6].

[1] http://monetdb.cwi.nl/xml
[2] http://dblp.uni-trier.de/xml
[3] http://pir.georgetown.edu
[4] http://www.nlm.nih.gov/bsd/pmresources.html
[5] http://alfred.med.yale.du/alfred
[6] http://www.tpc.org/tpch

Table 9.1: Document properties.

	Size (MB)	MaxDepth	VTags	Vattributes	VContent	VNSearch	#Tags	#Attributes	#Content	#NSearch
XMark1	55.32	12	148	9	85441	12	1665820	191160	9276986	13
XMark2	115.76	12	148	9	132359	12	3470166	397928	19384255	13
XMark3	513.96	12	148	9	417309	12	15381746	1762307	85916582	13
XMark4	1029.18	12	148	9	757852	12	30749422	3525025	171832697	13
Dblp2008	282.42	6	70	6	1750576	14	13856520	1426867	60222798	17
Dblp2012	961.75	6	70	9	4525940	14	47888064	6082270	214012325	17
Psd	683.64	7	128	7	3142459	9	42611636	1052770	105568992	9
Medline1	121.02	7	156	5	266168	0	5732160	138315	16490261	0
Medline2	593.14	7	164	15	894702	14	28478436	4436417	87413949	15
Medline3	877.32	7	166	16	1360745	14	40199504	6468566	131882636	15
Alfred	74.16	5	120	0	75630	14	4089784	0	8105935	17
Baseball	0.64	6	92	0	3149	0	56612	0	60897	0
Lineitem	30.80	3	36	1	39593	0	2045952	1	3411432	0
Mondial	1.78	5	46	32	19086	30	44846	47423	321201	33
Nasa	23.89	8	122	9	77687	0	953292	56317	4180538	0
Shakespeare	7.53	7	44	0	28346	9	359380	0	1505075	9
Swissprot	112.76	5	170	14	500909	0	5954062	2189859	23166916	0
Treebank	85.42	36	500	1	1979256	0	4875332	1	10439446	0
USHouse	0.51	16	86	21	5179	14	13424	2732	82414	15
Tcsd-normal	107.18	8	48	1	613408	33	5499502	7333	22129473	37
Scsd-normal	105.37	8	100	3	663514	33	4485398	150000	14547468	37
Uniprot1	434.99	6	144	39	1061320	14	17587730	11364588	89110893	15
Uniprot2	716.00	6	144	39	1608280	14	28999340	18671115	146563011	15
EXI-Array	22.06	10	94	17	94951	27	453046	226550	3600182	33
EXI-Factbook	4.04	5	398	0	28013	39	110906	0	604601	54
EXI-Invoice	0.93	7	104	7	16748	9	30150	14060	109538	9
EXI-Weblog	2.53	3	24	0	1260	0	186870	0	435894	0
EnwikiNews	69.42	5	40	7	311877	0	809304	35000	15416589	0
EnwikiQuote	124.27	5	40	7	412082	0	525910	23837	29155406	0
EnwikiTionary	556.61	5	40	7	3479730	0	16770268	726129	104853291	0
EnwikiVersity	81.40	5	40	7	300349	0	991678	43621	18830566	0
EnwikiAbstract1	660.56	5	18	1	540589	0	28327694	3811222	140817649	0
EnwikiAbstract2	327.96	5	18	1	420168	0	13692938	1714361	70280032	0

- `Mondial`: world geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources[7].

- `Nasa`: file from the *NASA XML Project* [8]. It contains astronomical datasets converted from legacy flat-file format into XML and then made available to the public.

- `Shakespeare`: file containing a collection of Shakespeare plays.

- `Swissprot`/`Uniprot`: manually and automatically annotated protein sequence databases[9] which provide a high level of annotations (such as the description

[7] http://www.cs.washington.edu/research/xmldatasets/www/repository.html
[8] http://xml.nasa.gov
[9] www.uniprot.org

of the function of a protein, its domains structure, post-translational modifications, variants, etc.).

- `Treebank`: file of parsed English sentences from the Wall Street Journal[10]. The main feature of this document is that all text nodes have been encrypted since they are copywritten text. It also shows a very deep and recursive structure.
- `USHouse`: legislative document that contains information about the ongoing work of the U.S. House of Representatives[11].
- `TCSD/DCSD`: documents belonging to the XBench family of benchmarks that capture different XML application characteristics[12]. The generated files are categorized as *text centric* (TC) or *data centric* (DC), depending on they contain data that are actually stored as XML (e.g. book collections in a digital library, and news article archives), or data which are not originally modeled in XML format (e.g. e-commerce catalog data and transactional data), respectively. These two models can be represented either in the form of a single document (SD) or multiple documents (MD). For our corpus, we selected TC-SD and DC-SD examples.
- `EXI`: sample documents from the Efficient XML Interchange (EXI) working group[13].
- `Wikipedia`: group of documents representing some extracted dumps from the English Wikipedia[14].

This collection of documents is used in Section 9.2 to evaluate the compression properties of our proposal, and to compare it with some other alternatives.

9.1.3 Query Test Bed

To benchmark the query evaluation performance of our tool (see Section 9.3), we have developed a complete query test bed for the `XMark` documents presented in Section 9.1.2[15]. The set of queries gives support to the whole practical subset of XPath discussed in Section 6.1, and aims to test the efficiency, scalability and stability of the analyzed systems. Queries are divided into four different categories as we will next describe:

[10] http://www.cis.upenn.edu/ treebank
[11] http://xml.house.gov
[12] http://www.cs.uwaterloo.ca/ tozsu/ddbms/projects/xbench/index.html
[13] http://www.w3.org/XML/EXI
[14] http://dumps.wikimedia.org/backup-index.html, http://dumps.wikimedia.org/enwiki
[15] We have focused on these documents of the data set, since the *XMark* project has been acknowledged as a reference for XML data benchmarking.

- `A (Q01-Q21)`: *XPathMark*[16] is a well established benchmark [Fra06] that provides a collection of queries to test the performance of an XML query processing system with regards to XPath 1.0. All the queries are intended to simulate realistic query needs of a potential user of an auction site modeled by any of the XML documents generated with the *xmlgen* tool of the *XMark* project (that is, the `XMark` documents of our corpus). Yet they are classified into several groups according to the fragment of XPath targeted (e.g. XPath axes, relational and arithmetic operators, positional functions, etc.).

  In this way, category `A` of our test bed takes the queries of the *XPathMark* benchmark related to the practical subset of XPath addressed in this work. In particular, all those groups of queries that cover the *forward* and *reverse* XPath axes, using as node tests either a tag/attribute name or the wildcard '`*`', and that admit the use of predicates, in combination with conjunctive and disjunctive boolean operators[17]. Indeed, we have also included some additional queries, created ad-hoc, exhibiting the same properties. They are all presented in Figure 9.1.

- `B (Q22-Q42)`: one of the most challenging scenarios for query evaluation is that posed by queries involving a sequence of steps over the wildcard '`*`', due to the potentially high number of intermediate results that can be generated (e.g. `/book/*/*//*/section`). This part of the query test bed is precisely devoted to validate the systems performance under these situations. Figure 9.2 shows the queries created to this aim, regarding both elements (Q22-Q32) and attributes (Q33-Q37). Additionally, queries from Q38 to Q42 constitute 'crash tests' specifically designed to work with various intermediate results sizes.

- `C (Q43-Q58)`: as users can be interested in selective queries, they may look for occurrences of specific elements and attributes, as well (e.g. `//book`, `//@reference`, etc.). This category focuses on this case. Selected queries of elements and attributes randomly chosen are shown in the left side of Figure 9.3. Notice that we also regard the special queries searching for any element (Q43) or attribute (Q54) appearance.

- `D (Q59-Q73)`: categories `A`, `B` and `C` do not consider text functions, as they are pure structural based queries. Hence, this last group has been devoted to cover examples of typical queries that an user could formulate over any `XMark` document, by using the `contains` or `equal` functions applied either over an element content (Q59-Q68) or even an attribute value (Q69-Q73). They have been created by considering both single words and phrase patterns, as shown in the right side of Figure 9.3.

[16] http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark

[17] Note that we do not consider relational and arithmetic operators, nor positional functions, as they are not addressed in this thesis.

Q01: /site/closed_auctions/closed_auction/annotation/description/text/keyword
Q02: //closed_auction//keyword
Q03: /site/closed_auctions/closed_auction//keyword
Q04: /site/closed_auctions/closed_auction[./annotation/description/text/keyword]/date
Q05: /site/closed_auctions/closed_auction[./descendant::keyword]/date
Q06: /site/people/person[./profile/gender and ./profile/age]/name
Q07: /site/people/person[./phone or ./homepage]/name
Q08: /site/people/person[./address and (./phone or ./homepage) and (./creditcard or ./profile)]/name
Q09: /site/regions/*/item[./parent::namerica or ./parent::samerica]/name
Q10: //keyword/ancestor::listitem/text/keyword
Q11: //happiness/ancestor::closed_auction/annotation/author
Q12: /site/open_auctions/open_auction/bidder[./following-sibling::bidder]
Q13: /site/*/person[./homepage/following-sibling::creditcard]/name
Q14: /site/open_auctions/open_auction/bidder[./preceding-sibling::bidder]
Q15: /site/people/person/*/gender[./preceding-sibling::education]
Q16: /site/regions/*/item[./following::item]/name
Q17: /site/open_auctions/open_auction/reserve/following::happiness
Q18: //type/preceding::price
Q19: /site/regions/*/item[./preceding::item]/name
Q20: //person[./profile/@income]/name
Q21: //open_auction[./privacy]/itemref/@item

Figure 9.1: First group of queries (A).

Q22: //mailbox/*/*/keyword
Q23: //namerica/*/mailbox//*/*/keyword
Q24: //open_auction/*/author
Q25: //regions/*/*/*/*/*/parlist//emph
Q26: //categories/*/description/*/*/keyword
Q27: //categories/*/description//*/*/keyword
Q28: //keyword/parent::*/parent::*/parent::mail/date
Q29: //author/parent::*/parent::open_auction/itemref
Q30: //parlist/parent::*/parent::*/parent::*/parent::*/parent::*/parent::regions
Q31: //keyword/parent::*/parent::*/ancestor::description/parent::category/name
Q32: //keyword/parent::*/ancestor::description/parent::item[./parent::namerica]/location
Q33: //open_auction[.//*/*/@person]/seller
Q34: //person[.//*/*/@category]/homepage
Q35: //person[./*/*/@open_auction]/name
Q36: //categories//*/@id
Q37: //person//*/@income
Q38: /*/*/*//*//*//*/*/*/*
Q39: /*/*/*/*/*/*/*/*/*
Q40: /*//*/*/*/*
Q41: /*/*/*/*
Q42: /*

Figure 9.2: Second group of queries (B).

Q43: //*
Q44: //edge
Q45: //australia
Q46: //province
Q47: //age
Q48: //street
Q49: //homepage
Q50: //parlist
Q51: //keyword
Q52: //date
Q53: //time
Q54: //@*
Q55: //@from
Q56: //@featured
Q57: //@income
Q58: //@id

Q59: //mail//text[contains(.,"image")]
Q60: //item/location[contains(.,"Island")]
Q61: //location[.="Israel"]
Q62: /site/regions/europe/*/location[.="United States"]
Q63: //open_auction/bidder[./date="09/13/1998"]
Q64: //payment[contains(.,"Creditcard")]
Q65: //australia//payment[contains(.,"Personal Check, Cash")]/parent::item/@id
Q66: //namerica//payment[contains(.,"Personal Check, Cash")]
Q67: //text[contains(.,"weaker dove")]
Q68: //annotation[contains(.,"dove miserable")]
Q69: //person/profile/@income[.="9876.00"]
Q70: /site/regions/*/item[./@featured.="yes"]/name
Q71: /site//interest[./@category="category266"]
Q72: //interest/@category[.="category328"]
Q73: //@category[.="category328"]

Figure 9.3: Third (C) and fourth (D) group of queries.

9.2 Compression Properties

XXS constitutes, in essence, a new XML queriable compression tool. Therefore, related to compression features, fair and consistent comparisons stand from its analysis against other queriable compressors. Yet, despite the large amount of research that has been developed along the years focused on this compression area, as stated in Section 4.2, almost all the tools presented in the literature do not have currently available source codes. To the best of our knowledge, solely the XGrind, XBzipIndex and SXSI tools are accessible. Of them, XGrind could not be run under the Linux version operating system of our test machine. Hence, just XBzipIndex and SXSI remain as available queriable compressors that have been benchmarked.

Even so, we have also decided to validate our proposal against some of the non-queriable compressors[18], as well as general text compression methods. Reader should notice that the comparison results in those scenarios can not be considered straightforward, since none of them exhibit the querying ability. They are shown just as basic references. Similarly to what happen with queriable compressors, XML conscious non-queriable tools also suffer from the lack of source code/binaries [Sak09]. As a result, only those available ones have been compared[19].

Our experimental environment includes the compressors next detailed. For any of the tested compressors, we use the maximum and minimum compression options whenever they exist:

[18] We discard schema-dependent compressors since they are not commonly used in practice.

[19] Apart from Exalt compressor. Although it is accesible, it failed to successfully compress most of the documents. Therefore, we excluded it from comparisons.

- **General text compressors**
  - `(s,c)-DC`: general back-end compression method used by the XWT representation.
  - `Plain Huffman`: another word-based byte-oriented semistatic statistical compressor, based on Huffman codes.
  - `Gzip`: a Ziv-Lempel based compressor. In particular, it makes use of the LZ77 technique. Fastest (-1) and best (-9) compression options of *gzip* are evaluated.
  - `Bzip2`: Seward's *bzip2*, a compressor based on the Burrows Wheeler Transform. As *gzip*, we also experiment with both the fastest (-1) and best (-9) alternatives.
  - `PPMdi`: as a representative method of the PPM family, we used PPMdi compressor, applying the minimum (-l 0) and maximum (-l 9) level of compression.
  - `p7zip`: is a LZMA based compressor with a dictionary of up to 4 Gigabytes.
- **XML conscious compressors**
  - *Non-queriable tools*
    - `XMill`: we have used the three general back-ends compressors provided by XMill, namely *gzip*, *bzip2* and *PPM*, thus yielding three different compressors: XMillGzip, XMillBzip2 and XMillPPM. Moreover, in case of XMillGzip, XMill allows one to set the compression factor to the minimum (-1) or maximum (-9) value.
    - `XMLPPM`: based on PPM compression scheme.
    - `SCMPPM`: the SCM variant achieving the highest compression ratios. It also supports fastest (-1) and best (-9) compression options.
    - `XWRT`: two variants are used, depending on we select *zlib*[20] or *lpaq*[21] as back-end compressors. Both alternatives provide maximum and minimum compression options. However, the compression gain obtained when using the maximum ones (less than 1%), does not pay off the compression times (between 1.5 and 2 times slower). Hence, minimum compression options are set when running these compressors.
  - *Queriable tools*
    - `XBzipIndex`: adaptation of the XML Burrows Wheeler Transform.

[20] It is based on the same *deflate* compression algorithm than *gzip* compressor (http://www.zlib.net/).

[21] http://mattmahoney.net/dc/

* SXSI: an up-to-date proposal for compressed indexing of XML documents.

Apart from compressors, we have also benchmarked XXS compression properties against some of the best current state of the art database based solutions supporting XPath, whose query performance will be further analyzed in Section 9.3. Both MonetDB/XQuery and Qizx/DB are the examples of systems from this category used there, that are included, as well, in this part of the study to validate them regarding their space features.

9.2.1 Results Evaluation

We have compared XXS with the above mentioned compressors and query systems. In case of pure compression methods, such as general text compressors and XML conscious yet non-queriable compressors, we have analyzed their main compression parameters, namely the compression ratio and the compression and decompression times. In turn, for actually queriable approaches, such as SXSI, MonetDB/XQuery, and Qizx/DB, we have measured the global size of the representation created to allow query evaluation, and also their construction times. Figures from 9.4 to 9.7 show the results obtained for each of the different XML documents previously described in Section 9.1. To allow a better understanding of these figures and the corresponding discussions, results are grouped by using different colour ranges, according to the following categorization of the solutions tested:

- *XXS*: the results obtained by our system are depicted in blue.
- *General text compressors*: they are all marked in black. We use $-f$ and $-b$ to make clear the distinction between the *fast* and *best* variants of a compressor, whenever these compression options are applicable.
- *XML conscious non-queriable compressors*[22]: in this case, results are highlighted by using the pink colour palette. Like general text compressors, we also use $-f$ and $-b$ options to mark the *fast* and *best* variants of some of these compression tools.

  It is important to note, as well, that some of the XML conscious compressors failed to either compress or decompress some of the documents. It is the case of XMill compressors, with regards to `Mondial`[23], or even of SCMPPM, related to that same document, but also to `Nasa`, `Uniprot` files and `Treebank`[24].

[22] We also include in this group the XBzipIndex. Although it is generally classified as a queriable XML conscious compressor, it provides a very limited query support in comparison with the rest of the queriable solutions.

[23] The resulting error is 'Parse error in line 15: Symbol '>' expected after '/' in tag!'.

[24] In case of `Treebank`, *fast* variant of SCMPPM does not fail, but the *best* one does. The error produced for all the failed documents arises during compression as 'Not well-formed document! <DL>'

Likewise, XBzipIndex and XWRT failed to compress `Dblp` documents[25] and to decompress `EXI-Factbook`[26], respectively.

- *Queriable solutions*: this group covers MonetDB/XQuery and Qizx/DB databases, but also SXSI tool. The values corresponding to these proposals are marked in green in Figure 9.4 and Figure 9.7. Notice that, in some cases, results are not shown for a given document due to system construction failures. For instance, MonetDB/XQuery and SXSI failed when working with `Dblp` documents and `Alfred` file[27]. The former did not succeed also over `USHouse`[28].

9.2.1.1 Compression Ratios

Figure 9.4 shows the compression ratios (in % with respect to the original document size) obtained by each of the compared solutions. Notice that regarding our proposal, we have distinguished two different compression ratios, marked as 'XWT' and 'XXS', respectively. Recall that XXS compressed storage arises from the XML Wavelet Tree data structure. Hence we denote with 'XWT' the compression ratios achieved by the XWT representation of each document, just considering the space needed to perform compression and decompression tasks. In turn, 'XXS' represents the waste of extra space needed for efficient query evaluation, including that used for the structures of partial counters to speed up *rank* and *select* operations over the XWT bytemaps (see Section 3.2.1.2), and also that needed for the succinct tree representation of the balanced parentheses data structure[29]. We include, as well, the amount of space used to maintain the vocabularies of words into hash tables. Notice that, in general, XXS space requirements amount an additional 4%-8% of extra space over the XWT basic representation[30]. In this way, 'XWT' values will be used for comparisons with general compression methods and XML conscious non-queriable compressors, whereas 'XXS' ones will be compared against queriable solutions.

XWT versus general text compressors. From the results presented in Figure 9.4 we can observe that, in general, XWT represents each document by using about 30%-40% of its original size. If we compare its performance against (s,c)-DC compressor, which constitutes the basis of XWT compression scheme, we will

[25] The compression fails trying to search for the document DTD.

[26] Output error: 'File corrupted (s.size()>WORD_MAX_SIZE)! Not enough memory!'.

[27] In case of `Dblp` files the failure arises when loading the external entity 'dblp.dtd'. For `Alfred` we obtain 'Parse error: space needed here <?xml version="1.0"? >'

[28] Output error: 'XML input not well-formed'.

[29] Recall that we use the fully-funcional succinct tree representation [SN10] presented in Section 3.2.2.

[30] There are some particular exceptions, such as `Mondial`, `Treebank`, `USHouse` and `EXI-Invoice`, for which these space differences are higher, mainly due to the amount of spaced needed to maintain the vocabulary hash tables.

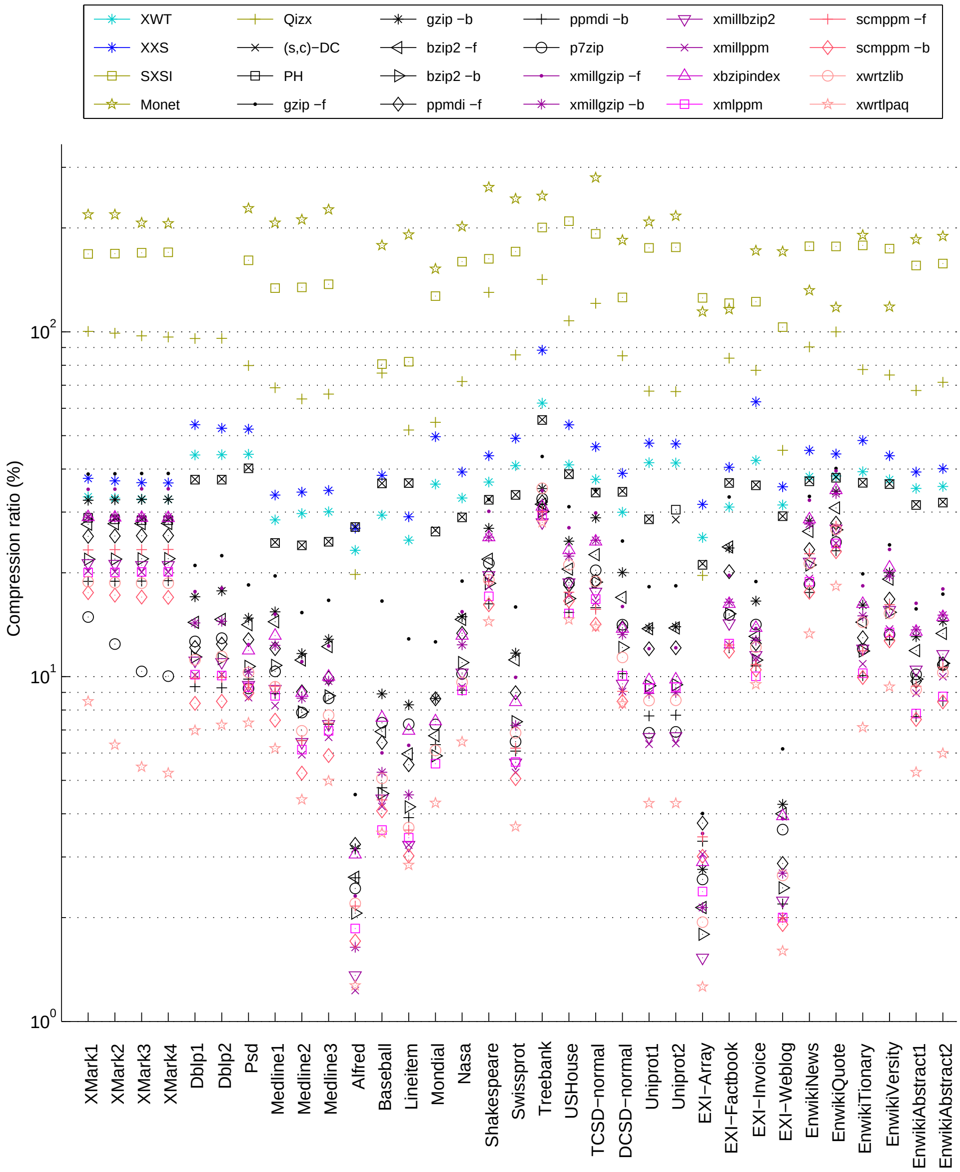


Figure 9.4: Compression ratios achieved by our proposal (in blue), general text compressors (in black), XML conscious non-queriable compressors (in pink), and queriable tools (in green) over different XML documents.

note that XWT needs, on average, just about 3%-4% more space than (s,c)-DC to compress the same document. However, within such a little difference, XWT exhibits some properties that are key to further allow XML querying purposes. Note as well, that a similar remark can be done if we consider the other example of word-based byte-oriented compressor used, that is, Plain Huffman.

In comparison with the rest of the general text compressors, the compression ratios achieved by the aforementioned techniques (that is, XWT, (s,c)-DC and Plain Huffman) are higher, as expected. In this case, the reader should have in mind that (s,c)-DC (and by extension also XWT) and Plain Huffman are mostly intended to compress natural language text. In fact, one can notice that for the documents close to that nature[31] (such as `XMark` files, `Shakespeare`, `TCSD`, `DCSD`, `EXI-Factbook`, `EnwikiNews`, and `EnwikiQuote`) differences are not as significant.

On the other hand, if we just focus on the comparison among the general text compressors themselves, apart from XWT, (s,c)-DC and Plain Huffman, we can observe that, in general, *gzip* variants obtain the worst compression ratios, while *bzip*, *ppmdi* and *p7zip* show a quite similar performance. Yet the best variant of *ppmdi* usually achieves the best compression ratio for each document.

XWT versus XML conscious non-queriable compressors. The compression ratios of most of the XML conscious compressors tested are closely related to that of the corresponding general back-end compressors (such as *gzip*, *bzip2*, and PPM variants). Therefore, similar conclusions to those disclosed from the comparisons between XWT and the general compressors can be inferred also for this scenario. As it can be noted in Figure 9.4, XWT obtains worse compression ratios than the rest of the XML conscious non-queriable compressors. However, reader must recall that the tools from this category precisely aim to compress to the best, rather than equally provide an efficient query support, as they do not admit any query ability[32].

If we analyze the performance of XML conscious non-queriable compressors among themselves, we can observe that, in general, *gzip* based compressors, such as XMillGzip variants, and also XBzipIndex, are overcome by *bzip2* variants, like XMillBzip2, which is in turn beaten by PPM based alternatives, as XMillPPM, XMLPPM and SCMPPM compressors. Now going into detail, XBzipIndex behaves quite similarly to the best variant of XMillGzip. The same happens to XMillPPM with regards to XMLPPM. However, in case of SCMPPM, the fast compression option achieves results which are much closer to that obtained by XMillBzip2, than that of PPM based compressors. Yet, when using the maximum compression option, SCMPPM performs better in terms of compression ratio, than any of them.

[31] That is, with a larger number of natural language text fragments.

[32] Apart from XBzipIndex. Yet, even XBzipIndex can not be actually considered a queriable tool, as previously pointed out.

It is worth noting, as well, the behavior of XWRT, since XWRTzlib compresses, in general, better than the rest of the *gzip* based compressors[33], and closer to XMillBzip2, whereas XWRTlpaq is by far the XML conscious compressor which obtains the best compression ratios.

XXS versus queriable solutions. To perform the space comparisons against the queriable solutions, we have considered the overall space usage of our proposal, including the amount of extra space needed to speed up *rank* and *select* operations over the XWT bytemaps, as well as the corresponding counterpart for the balanced parentheses data structure, and also the space waste of maintaining the vocabularies into hash tables. Recall that these values are represented in Figure 9.4 under the 'XXS' label. As it can be observed, our proposal is by far the system that obtains the best compression ratios, followed by Qizx/DB, SXSI, and finally, MonetDB/XQuery, whose space requirements rise up to twice the original document size for almost all the tested files. As a queriable compression tool, based on a compressed and self-indexed representation of the document, XXS uses an amount of space closer to that obtained by pure compression methods. However, the most remarkable feature, is that within such a little amount of space, XXS is able to provide powerful XPath evaluation capabilities, like the queriable solutions that require, on average, between 2 and 5 times more space than our solution.

9.2.1.2 Time Measures

Regarding time measures, we next analyze the compression and decompression times (in seconds) of the different compressors tested (see Figure 9.5 and Figure 9.6) and also the construction times (in seconds, as well) of the queriable solutions (see Figure 9.7). For XXS, we must note that construction times are actually given by the time required to compress the document, that is, to create the XWT representation and to store it into disk, since the additional *rank*/*select* structures used for efficient searching are created on-the-fly when data structures are loaded from disk.

XWT versus general text compressors. As depicted on top of Figure 9.5, if we compare XWT against (s,c)-DC and Plain Huffman codes, we will note that XWT takes larger times to compress the input data, due to the more complex parsing we perform to meet XML features. In turn, decompression times are not affected. What is more, they are even improved in many cases (see the graph on top of Figure 9.6).

From the behavior of the rest of the general text compressors, we can infer that XWT outperforms both compression and decompression times of virtually all of them. Just in case of the fast variant of *gzip*, this compressor obtains better compression times than XWT (and actually than (s,c)-DC and Plain Huffman) for most of the documents. If we consider the *gzip* best variant that conclusion is not

[33] Recall that both *zlib* and *gzip* are based on the same `deflate` compression algorithm.

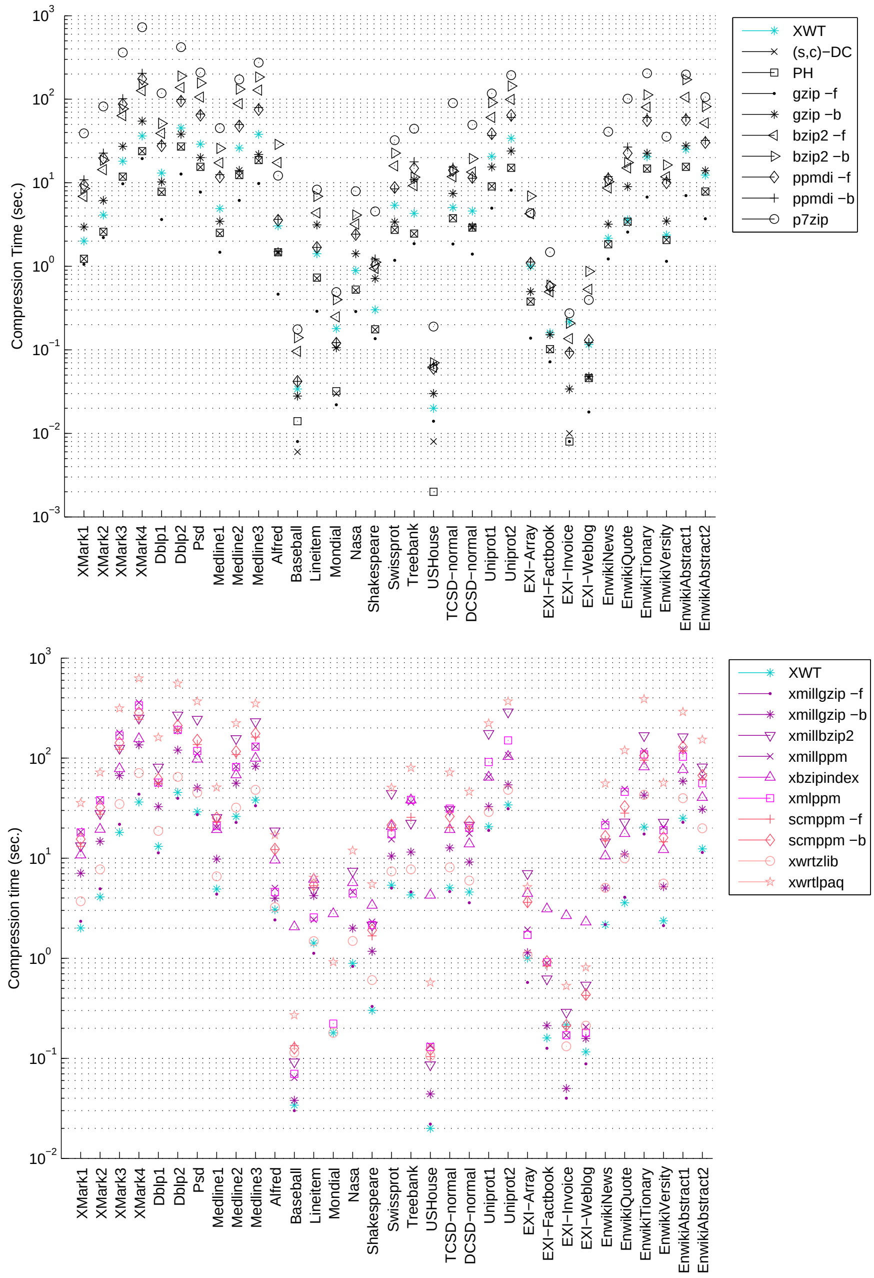


Figure 9.5: Compression times. Comparison with general text compressors (top), and with XML conscious non-queriable compression tools (bottom).

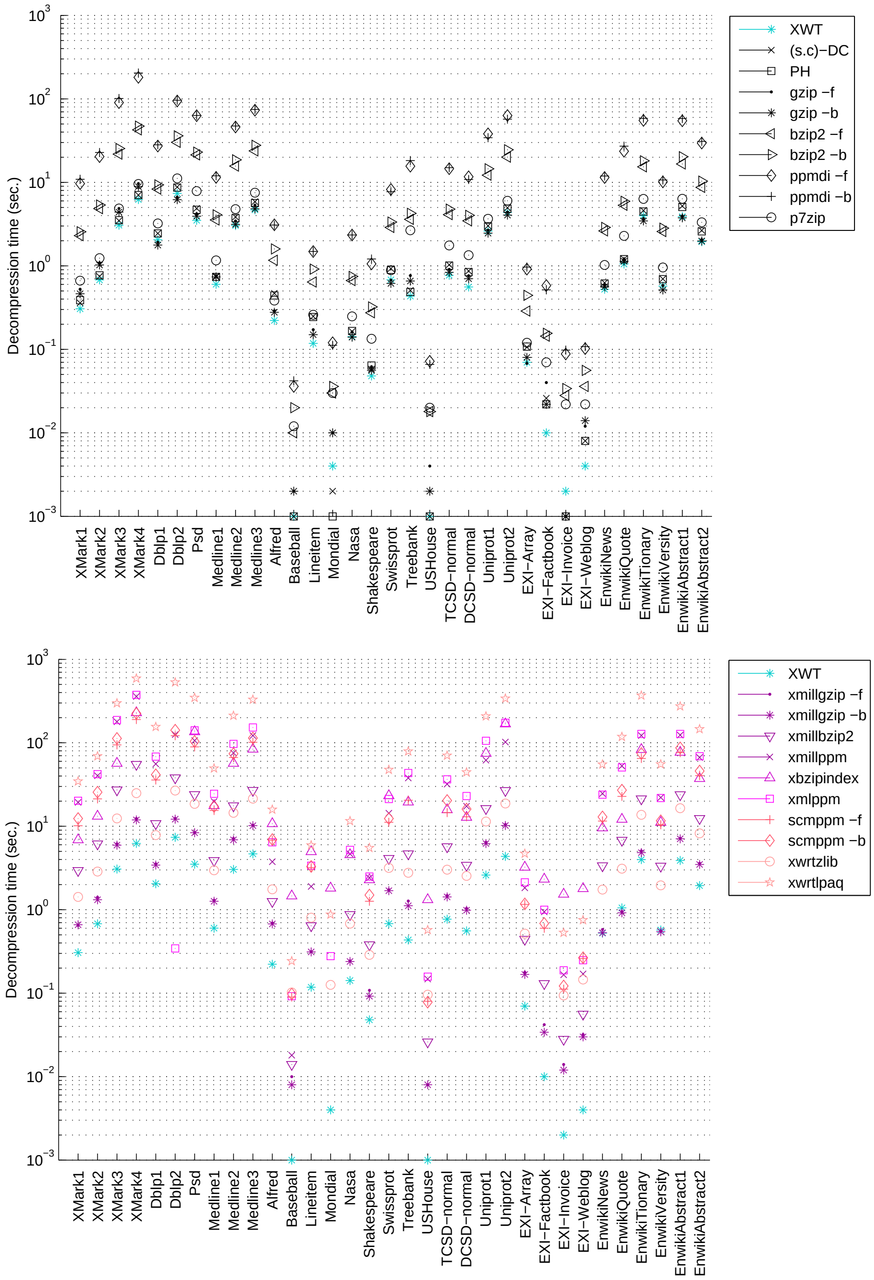


Figure 9.6: Decompression times. Comparison with general text compressors (top), and XML conscious non-queriable compressors (bottom).

as clear, as XWT obtains lower times for some documents. Yet, regarding decompression times, both variants of *gzip* are overcome by XWT.

Related to the performance of the remaining general compressors, that is, *bzip* compressors, and *ppmdi* and *p7zip* tools, the performance of *p7zip*, particularly slow in compression, but not at decompression, is remarkable. In this last scenario, *ppmdi* compressors have the longest decompression times, followed by *bzip* techniques, and finally by *p7zip*[34], whose results are far from the high decompression times of the previous ones, and much closer to that of the fastest methods.

XWT versus XML conscious non-queriable compressors. Regarding the times invested to compress the documents (see the graphic at the bottom of Figure 9.5), XWT achieves almost the same compression times as XMillGzip compressor with the minimum compression options, both tools being the best ones. The rest of the techniques are largely slower. Actually, they are much slower than their general counterparts, so much that compression ratios improvements are ultimately blurred by time requirements. Just XWRTzlib is able to achieve compression ratios comparable to that of *p7zip*, but in less time.

From a detailed analysis, we can observe that PPM based techniques work similarly, all of them showing worst compression times than *gzip* based XML conscious compressors, and in general, than XBzipIndex. Yet the times obtained by XBzipIndex are much closer to those required by PPM based compressors, than to XMillGzip or XWRTzlib values. Notice, as well, that whereas XWRTlpaq was the compressor with the best compression ratios, now it is the one with the largest compression times. Also XMillBzip2 is quite slow, even worsening the results of PPM based XML conscious compressors.

The graphic at the bottom of Figure 9.6 represents the differences of decompression times among the XML conscious non-queriable compressors. In this case, XWT has no competitor. Although far from XWT, the next compressors requiring lower decompression times are both variants of XMillGzip, followed by XWRTzlib. Regarding XBzipIndex, it produces again higher decompression times than XMillGzip and XWRTzlib compressors, in particular, similarly to that required by SCMPPM compressors. SCMPPM alternatives constitute the best of the PPM based techniques, since both XMillPPM and XMLPPM are slower. Anyway, XWRTlpaq yields once more the worst decompression times. If we focus on XMillBzip2, a change on its behavior can be observed. This compressor required high compression times, but at decompression, it performs close to, although not better than, XWRTzlib.

XXS versus queriable solutions. If we stare at the construction times of the queriable solutions shown in Figure 9.7, we will note that XXS and MonetDB/XQuery are the fastest alternatives, both requiring only a few seconds, while

[34] Contrary to what happened in case of compression times.

Qizx/DB and SXSI may take several minutes. In particular, SXSI construction times are specially slow.

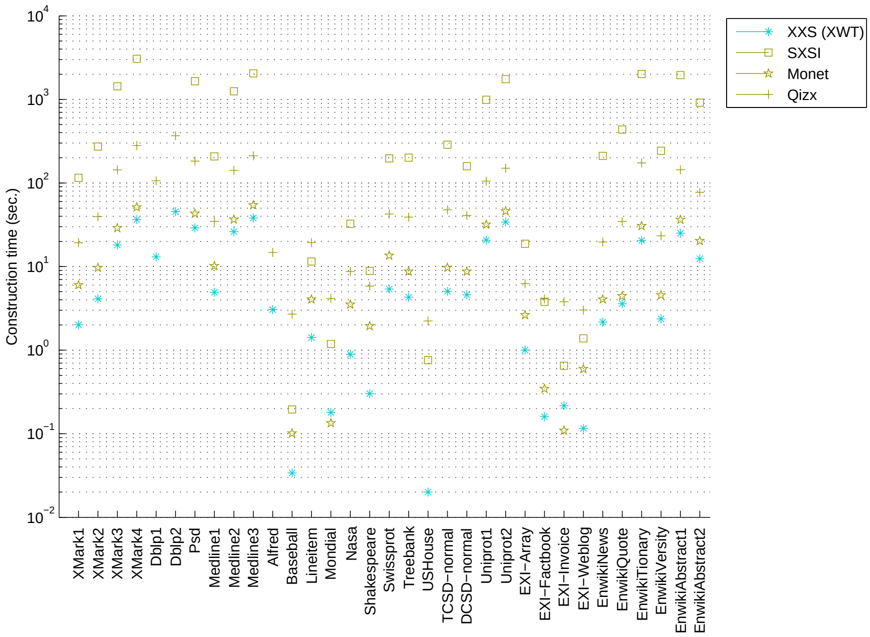


Figure 9.7: Construction times of queriable solutions.

9.3 Query Evaluation Performance

To show the efficiency of XXS in query evaluation, we have benchmarked it against the performance of some queriable solutions. In particular, those first referred in Section 9.2, namely MonetDB/XQuery[35], Qizx/DB[36], and SXSI. All of them constitute well established solutions from the different categories analyzed in Chapter 4.

Although XBzipIndex is generally categorized as a queriable compression tool, we have discarded it from this study, as it gives support to a very limited class of XPath queries[37]. It is also worth noting that we have decided to not go in depth

[35] We used version Oct2010-SP1 of MonetDB, that includes version 4.40.3 of MonetDB4 server and version 0.40.3 of the XQuery module

[36] We used Qizx/DB free edition, version 4.2.

[37] Remember that only full-specified paths of the form $//x_1/x_2/x_3$ and $//x_1/x_2/x_3[contains(.,\gamma)]$, where x_1, x_2 and x_3 denote tag/attribute names, and γ is an arbitrary string, are supported by XBzipIndex.

on the streaming XPath engines (e.g. GCX, and SPEX), or on the in-memory processors (e.g. Galax, and SAXON). Such a comparison is hardly fair, since in the first case, streaming processors need to *parse* the whole XML document input at each run. For instance, when compared with XXS, SPEX streaming processor runs about 475 times slower than our proposal. In turn, the limitation of in-memory processors comes from the construction times required to build the in-memory representation, prior to evaluating the query at each run, which in case of SAXON, made it run about 125 times slower than XXS. In addition, this kind of tools usually represents XML data by means of machine pointers implementations that blow up memory consumption. In particular, SAXON needs 4-5 times the size of the original XML documents used in our experiments.

9.3.1 Documents Tested

As first pointed out in Section 9.1, the experimental framework for query evaluation has been designed to be tested over any `XMark` document. In particular, we have run the experiments over the `XMark2` and `XMark4` documents of our collection. Table 9.2 details the construction features (taken from the results of Section 9.2) of the different systems over those selected documents, namely, the size of the data structures created to perform query evaluation (in % of the document size), and also the construction times (in seconds). We marked in boldface the best values. In case of XXS, the results presented correspond to a XWT implementation with a particular waste of 3% of extra space for the space needed to build the *rank/select* structures over the XWT bytemaps, and also for the succinct representation of the balanced parentheses data structure[38].

Table 9.2: Systems construction performance.

	Document	*Total size (%)*				*Construction time (sec.)*			
	size (MB)	XXS	SXSI	Monet	Qizx	XXS	SXSI	Monet	Qizx
XMark2	115.76	**36.94**	168.58	218.73	99.05	**4.11**	272.64	9.66	39.55
XMark4	1029.18	**36.43**	169.98	206.03	96.54	**36.37**	3059.26	51.11	280.50

9.3.2 Query Results

To perform the query evaluation tests we kept the best of five runs, for each query, by using the systems timing reports. For MonetDB, times are given by the *-t* option of the client program, `mclient`. The server is properly exited and restarted before each group of five runs. For Qizx/DB, we used the $-r$ 2 option of the command

[38]The space needed to maintain the vocabularies into hash tables represents around 1% of the compression ratios shown in Table 9.2.

line interface to run twice each individual run (the second one, being always faster). For all systems, we do not consider the index loading times into main memory.

9.3.2.1 Structural based Queries

We first addressed the performance of the four systems with respect to the evaluation of structural based queries, represented by the groups of queries `A` (Q01-Q21), `B` (Q22-Q42) and `C` (Q43-Q58) presented in Section 9.1.3. Tables from 9.3 to 9.8 summarize, for each individual query, the running times (in milliseconds) of the main search operations: *count*, *materialize*, and *materialize+serialize* the results[39]. In the first scenario, queries shown in Section 9.1.3 are run by adding the XPath `count` function to each one. For instance, a query such as `//closed_auction //keyword` will result into `count(//closed_auction//keyword)`. The results materialization stands for their location, while the third operation also includes the actual results display. For Qizx/DB, it is not possible to isolate the materialization times, so we only compare it in the other two scenarios. Notice also, that some of the queries could not be run by SXSI, since it does not support `following`, `attribute`, nor reverse axes.

In general, we can conclude that XXS shows an outstanding performance, specially if we consider that its competitors require $2-5$ times more space than our proposal. At first sight, we can observe that it is the only system that reached to solve all the queries.

Related to counting and materializing scenarios, timing results show that XXS performs on par with the other solutions and even better, since it achieves the best running times in most queries. It is also important to notice that in those cases, XXS and SXSI do not experience performance variability with the document size (in terms of number of queries reporting the best running times). Nevertheless, it is not the case of MonetDB/XQuery or Qizx/DB. The former performs better over `XMark2`, but its performance gets worse over `XMark4`; while the opposite happens to Qizx/DB, specially on the group of queries `A`. Anyway, Qizx/DB does not show, in general, a remarkable performance, as it gets the best running times in very few cases.

With respect to materializing plus serializing times, most of the best results are obtained by MonetDB/XQuery and SXSI, when dealing with the small document instance. However, MonetDB/XQuery does not get as good timing results for the biggest document (`XMark4`), while XXS and SXSI still do. Regarding Qizx/DB, again it does not get any outstanding result. To properly valuate the XXS performance in this scenario[40], we have to consider that one of the main advantages of our proposal are its minimal space requirements. It works over a compressed

[39] Results are serialized to the `/dev/null` device, to discard data output.

[40] Note that, in this case, XXS times taken to serialize the results obfuscate the pure query processing time.

(and self-indexed) version of the text, and with no additional indexes to keep the compression gain. Therefore, to fully report the results of a query we have to decompress each word by decoding it through top-down traversals over the XWT, while in SXSI, for instance, a plain text representation is used to precisely enable fast text extraction.

In addition, since another main feature of XXS is the possibility of obtaining the results on user demand, we also consider important to show its behavior by assuming a scenario where the results are gradually consumed. Hence, we show the execution times of reporting the first 50 results for each query, as well (see data marked in blue from Table 9.3 to Table 9.8). Note that in most cases, those results are reported in less than one millisecond.

The aforementioned conclusions applies to the three groups of queries analyzed in this section. However, it is worth also to discuss the performance of XXS by focusing on groups `B` and `C`, as both constitute special groups of queries:

- In case of queries of group `B`, Table 9.5 and Table 9.6 illustrate the great performance of our solution, in particular, the efficiency of the parameterized operators (such as $\texttt{descendant}_{\texttt{dist}}$, $\texttt{child}_{\texttt{dist}}$, $\texttt{ancestor}_{\texttt{dist}}$, $\texttt{parent}_{\texttt{dist}}$, and their respective `att` counterparts), designed to overcome the challenge posed by the evaluation of queries with several steps over the '`*`' wildcard. Even in case of 'crash tests' (see queries Q38-Q42) XXS shows its robustness, as well.

- On the other hand, and regarding group `C` (see Table 9.7 and Table 9.8), we should recall that these queries aim to seek for all the occurrences of a given element/attribute (also including the special node test '`*`', in queries Q43 and Q54). Unlike other cases, where XXS times to count and to materialize the results are the same, in this group, time results for counting scenario benefit from the optimization explained in Section 8.2.2.15. Recall that, as stated there, for this kind of queries XXS simply needs to compute how many times the last byte of the codeword assigned to the specific element/attribute appears in its corresponding XWT node. As shown in Tables 9.7 and 9.8, this operation is performed very efficiently, just requiring some microseconds, beating by far the rest of the systems.

  Another interesting feature that arises from the analysis of this group of queries is that, in case of materialization times of element searches, MonetDB/XQuery performs particularly well even over the biggest document instance. Yet, it does not cope with attributes, while XXS does.

9.3.2.2 Text oriented Queries

To evaluate the performance of the systems over queries involving a text function, we used the *Full text* extension of XQuery [Ful] available in the tested version of

Table 9.3: Running times (in milliseconds) for the group of queries `A` over `XMark2` document.

	XMark2												
	Count				*Materialize*				*Materialize+Serialize*				
	XXS_{all}	SXSI	Monet	Qizx	XXS_{50}	XXS_{all}	SXSI	Monet	XXS_{50}	XXS_{all}	SXSI	Monet	Qizx
Q01	19.91	**14.86**	17.46	26.00	0.307	19.91	**15.39**	17.70	1.208	87.83	**20.95**	23.99	66.00
Q02	**7.41**	9.51	8.56	42.00	0.030	**7.41**	14.20	9.13	0.725	202.17	30.54	**24.52**	54.00
Q03	**8.13**	9.23	10.99	14.00	0.036	**8.13**	13.86	11.36	0.725	203.04	30.24	**28.18**	47.00
Q04	21.89	**17.76**	21.45	19.00	0.546	21.89	**18.35**	21.65	0.725	32.61	**20.99**	25.96	74.00
Q05	**10.01**	23.43	16.66	22.00	0.095	**10.01**	24.44	16.82	0.290	27.39	30.00	**24.19**	73.00
Q06	27.24	30.18	23.65	**18.00**	0.463	27.24	31.56	**23.83**	0.966	62.61	35.30	**28.80**	182.00
Q07	33.08	**25.24**	37.63	76.00	0.115	33.08	**27.06**	37.21	0.531	217.83	**45.98**	54.49	93.00
Q08	36.97	**32.38**	49.13	79.00	0.280	36.97	**33.96**	49.27	0.725	108.70	**41.53**	57.53	118.00
Q09	**12.79**	*	52.38	58.00	0.116	**12.79**	*	53.12	0.483	104.78	*	68.01	**61.00**
Q10	91.15	*	**44.71**	258.00	0.154	91.15	*	**43.68**	0.773	600.37	*	**85.45**	285.00
Q11	**16.42**	*	20.33	115.00	0.101	**16.42**	*	20.87	0.580	107.83	*	**29.32**	112.00
Q12	58.84	**23.15**	31.66	79.00	0.099	58.84	**28.51**	32.68	0.628	565.22	**110.48**	181.53	309.00
Q13	21.73	25.51	**20.74**	72.00	0.193	21.73	26.83	**20.99**	0.580	86.96	33.86	**27.65**	112.00
Q14	**45.82**	*	138.53	206.00	0.087	**45.82**	*	139.29	0.628	550.87	*	**288.24**	448.00
Q15	**11.17**	*	27.97	35.00	0.182	**11.17**	*	28.04	0.290	**22.61**	*	31.00	79.00
Q16	**24.23**	*	+	476.00	0.057	**24.23**	*	+	0.435	**209.13**	*	+	547.00
Q17	**10.56**	*	33.33	+	0.054	**10.56**	*	33.55	0.242	84.35	*	**54.78**	+
Q18	**3.93**	*	54.71	+	0.029	**3.93**	*	56.69	0.338	**56.52**	*	67.37	+
Q19	**24.25**	*	+	+	0.058	**24.25**	*	+	0.435	**209.13**	*	+	+
Q20	42.28	*	**28.61**	65.00	0.226	42.28	*	**28.48**	0.676	168.70	*	**41.51**	109.00
Q21	35.73	*	**24.40**	96.00	0.367	35.73	*	**24.03**	0.480	50.00	*	**27.40**	114.00
$\star$: Query with axes not supported by SXSI +: Query did not finish													
Fastest running times are marked in boldface													

Table 9.4: Running times (in milliseconds) for the group of queries `A` over `XMark4` document.

	XMark4												
	Count				*Materialize*				*Materialize+Serialize*				
	XXS_{all}	SXSI	Monet	Qizx	XXS_{50}	XXS_{all}	SXSI	Monet	XXS_{50}	XXS_{all}	SXSI	Monet	Qizx
Q01	179.44	**111.07**	199.18	175.00	0.349	179.44	**114.41**	213.20	1.215	786.92	**164.82**	1453.99	294.00
Q02	67.66	76.51	164.76	**64.00**	0.059	**67.66**	117.12	168.53	0.748	1803.85	**263.48**	1393.73	557.00
Q03	73.93	70.16	174.79	**66.00**	0.073	**73.93**	111.89	181.78	0.934	1809.23	**258.05**	1368.61	565.00
Q04	197.38	134.05	217.74	**123.00**	0.576	197.38	**137.78**	221.34	0.841	293.85	**162.49**	1430.75	187.00
Q05	**90.28**	187.49	202.30	174.00	0.138	**90.28**	194.58	200.78	0.280	250.77	**244.67**	1388.79	381.00
Q06	243.18	242.13	347.47	**106.00**	0.461	**243.18**	252.29	345.61	0.934	564.62	**284.73**	1035.76	537.00
Q07	301.22	**201.67**	449.95	444.00	0.124	301.22	**214.97**	439.04	0.561	2006.15	**384.61**	1141.98	613.00
Q08	334.30	**251.14**	488.25	537.00	0.299	334.30	**261.61**	488.04	0.654	1003.08	**329.61**	1154.95	592.00
Q09	**114.49**	*	1017.51	3946.00	0.077	**114.49**	*	1015.49	0.561	**954.67**	*	3476.10	4193.00
Q10	**837.10**	*	2788.87	12584.00	0.156	**837.10**	*	2779.43	0.748	**5599.23**	*	8717.87	18649.00
Q11	**148.04**	*	1290.47	3428.00	0.148	**148.04**	*	1325.94	0.561	**978.67**	*	1407.37	4488.00
Q12	538.32	**185.41**	1078.08	669.00	0.095	538.32	**231.92**	1125.28	0.561	5191.08	**977.32**	2614.23	3768.00
Q13	**197.66**	202.28	619.44	458.00	0.211	**197.66**	212.27	621.47	0.654	800.00	**274.70**	880.73	552.00
Q14	**417.85**	*	2079.71	2163.00	0.085	**417.85**	*	2096.99	0.561	5062.06	*	**3622.60**	4381.00
Q15	**99.72**	*	684.28	418.00	0.191	**99.72**	*	682.57	0.280	**206.92**	*	955.96	374.00
Q16	**216.64**	*	+	15117.00	0.061	**216.64**	*	+	0.374	**1879.23**	*	+	15499.00
Q17	**95.51**	*	1324.84	+	0.049	**95.51**	*	1436.23	0.280	**727.69**	*	3826.42	+
Q18	**35.33**	*	2989.78	+	0.037	**35.33**	*	3032.29	0.187	**503.85**	*	3969.07	+
Q19	**217.01**	*	+	+	0.055	**217.01**	*	+	0.467	**1876.92**	*	+	+
Q20	388.41	*	**377.44**	414.00	0.215	**388.41**	*	396.95	0.654	1545.38	*	1127.75	**569.00**
Q21	**323.93**	*	357.78	358.00	0.349	**323.93**	*	364.25	0.467	452.31	*	**391.42**	459.00
$\star$: Query with axes not supported by SXSI +: Query did not finish													
Fastest running times are marked in boldface													

Table 9.5: Running times (in milliseconds) for the group of queries `B` over `XMark2` document.

	XMark2												
	Count				*Materialize*				*Materialize+Serialize*				
	XXS_{all}	SXSI	Monet	Qizx	XXS_{50}	XXS_{all}	SXSI	Monet	XXS_{50}	XXS_{all}	SXSI	Monet	Qizx
Q22	**17.39**	44.11	19.67	42.000	0.060	**17.39**	47.15	18.58	0.821	235.22	64.49	**38.34**	78.00
Q23	**9.15**	53.93	21.67	13.00	0.113	**9.15**	56.93	23.60	0.773	120.87	66.38	**33.60**	62.00
Q24	**14.18**	91.51	20.97	56.00	0.081	**14.18**	96.32	19.93	0.580	130.44	110.52	**34.46**	100.00
Q25	**16.92**	124.59	40.52	72.00	0.106	**16.92**	127.57	41.55	0.821	133.48	136.96	**52.66**	68.00
Q26	**0.71**	5.28	11.52	13.00	0.589	**0.71**	5.44	11.15	1.450	**1.30**	5.57	11.39	13.00
Q27	**0.82**	10.35	11.86	11.00	0.044	**0.82**	10.84	12.05	0.773	14.78	**12.13**	13.33	40.00
Q28	**18.72**	*	47.49	149.00	0.596	**18.72**	*	47.39	0.966	**29.13**	*	50.47	171.00
Q29	**21.50**	*	21.78	112.00	0.122	**21.50**	*	21.61	0.531	135.22	*	**34.36**	96.00
Q30	**0.003**	*	34.77	105.00	0.003	**0.003**	*	34.73	539.370	541.48	*	**488.00**	654.00
Q31	**1.06**	*	57.37	159.00	0.176	**1.06**	*	56.93	0.628	**3.48**	*	57.51	165.00
Q32	**18.04**	*	59.41	216.00	0.287	**18.04**	*	58.92	0.531	**43.48**	*	65.09	231.00
Q33	**55.80**	*	145.70	561.00	0.290	**55.80**	*	145.51	0.773	171.74	*	**159.95**	578.00
Q34	**29.36**	*	91.99	318.00	0.316	**29.36**	*	91.82	0.725	**81.74**	*	99.24	374.00
Q35	**34.97**	*	82.50	141.00	0.192	**34.97**	*	82.33	0.580	146.96	*	**94.03**	162.00
Q36	**1.53**	*	8.62	17.00	0.113	**1.53**	*	9.13	0.193	**3.91**	*	9.56	28.00
Q37	**22.71**	*	64.07	196.00	0.131	**22.71**	*	62.69	0.241	**51.74**	*	67.87	226.00
Q38	**28.73**	325.73	261.46	5403.00	0.009	**28.73**	372.63	259.35	1.014	5414.35	783.89	**524.79**	6298.00
Q39	**13.75**	204.06	109.40	683.00	0.007	**13.75**	222.98	107.57	1.400	2836.96	437.07	**271.05**	1150.00
Q40	**209.24**	498.38	333.26	2515.00	0.009	**209.24**	664.70	325.37	0.241	33666.10	3999.33	**2448.67**	9567.00
Q41	63.88	68.43	**31.61**	149.00	0.012	63.88	109.69	**30.58**	7.440	9463.48	1261.67	**1198.04**	3089.00
Q42	**0.009**	0.88	5.49	1.00	0.001	**0.009**	0.92	4.60	1051.400	1117.83	**867.63**	1187.31	3221.00

⋆: Query with axes not supported by SXSI   Fastest running times are marked in boldface

Table 9.6: Running times (in milliseconds) for the group of queries `B` over `XMark4` document.

	XMark4												
	Count				*Materialize*				*Materialize+Serialize*				
	XXS_{all}	SXSI	Monet	Qizx	XXS_{50}	XXS_{all}	SXSI	Monet	XXS_{50}	XXS_{all}	SXSI	Monet	Qizx
Q22	**158.97**	372.25	552.21	353.00	0.059	**158.97**	395.45	607.06	0.748	2067.33	**550.15**	5542.68	1585.00
Q23	81.96	429.90	298.68	**75.00**	0.075	**81.96**	453.87	318.36	0.934	1053.85	534.53	2506.55	**376.00**
Q24	**127.20**	796.12	526.44	290.00	0.073	**127.20**	857.33	524.44	0.561	1188.77	**988.70**	1009.86	1170.00
Q25	**152.80**	1066.80	723.32	259.00	0.106	**152.80**	1091.68	801.17	0.841	1210.00	1179.68	5865.52	**631.00**
Q26	**6.08**	20.51	30.90	70.00	0.706	**6.08**	20.92	35.31	1.495	**14.62**	21.49	137.89	85.00
Q27	**6.82**	45.77	37.74	72.00	0.061	**6.82**	48.34	37.81	0.654	128.46	**59.01**	145.77	115.00
Q28	**170.00**	*	2620.16	11674.00	0.603	**170.00**	*	2725.06	0.841	**265.39**	*	6134.37	11771.00
Q29	**193.08**	*	1277.29	4242.00	0.108	**193.08**	*	1280.66	0.467	**1213.08**	*	1400.22	5126.00
Q30	**0.006**	*	1471.50	4391.00	0.006	**0.006**	*	1452.63	5004.530	**4983.46**	*	6657.49	25138.00
Q31	**9.07**	*	2816.36	12764.00	0.211	**9.07**	*	2834.58	0.654	**33.08**	*	2806.94	12828.00
Q32	**162.52**	*	2812.76	13420.00	0.197	**162.52**	*	2819.11	0.467	**398.46**	*	4348.52	14712.00
Q33	**508.13**	*	1604.71	6630.00	0.282	**508.13**	*	1586.22	0.841	**1558.46**	*	2058.12	7665.00
Q34	**267.85**	*	934.28	2453.00	0.314	**267.85**	*	935.80	0.748	**746.15**	*	1632.58	2573.00
Q35	**320.75**	*	880.27	1213.00	0.225	**320.75**	*	875.88	0.561	1342.31	*	1560.14	**1319.00**
Q36	**12.71**	*	32.03	92.00	0.150	**12.71**	*	34.27	0.280	**33.08**	*	36.83	127.00
Q37	**206.36**	*	702.54	1471.00	0.124	**206.36**	*	699.20	0.187	**459.23**	*	742.30	1536.00
Q38	**259.72**	2842.77	3622.11	61387.00	0.009	**259.72**	3336.40	3620.15	1.121	47665.20	**7432.12**	12500.89	76102.00
Q39	**125.05**	1771.12	2352.33	20521.00	0.008	**125.05**	1945.51	2347.83	1.402	24987.90	**3868.97**	11706.21	23397.00
Q40	**1863.08**	4034.35	4050.25	44569.00	0.008	**1863.08**	6669.74	4101.93	0.280	296303.00	36412.75	**24665.71**	124960.00
Q41	**563.93**	589.70	1572.09	11242.00	0.012	**563.93**	950.93	1604.03	7.196	84443.10	**11249.80**	14584.12	41609.00
Q42	**0.002**	0.87	7.70	1.00	0.002	**0.002**	0.90	13.59	10071.700	10117.00	**7708.88**	15705.40	43168.00

⋆: Query with axes not supported by SXSI   Fastest running times are marked in boldface

Table 9.7: Running times (in milliseconds) for the group of queries `C` over `XMark2` document.

	XMark2													
	Count				*Materialize*				*Materialize+Serialize*					
	XXS_{all}	SXSI	Monet	Qizx	XXS_{50}	XXS_{all}	SXSI	Monet	XXS_{50}	XXS_{all}	SXSI	Monet	Qizx
Q43	**0.001**	0.684	38.25	618.00	0.008	224.78	171.15	**30.24**	1602.500	49428.10	**6398.10**	6596.96	19873.00
Q44	**0.003**	0.580	5.78	3.00	0.017	**0.16**	2.33	6.17	0.097	8.26	**3.46**	7.55	30.00
Q45	**0.003**	0.585	4.85	1.00	0.003	**0.009**	0.629	4.92	55.507	57.39	**45.10**	51.59	160.00
Q46	**0.003**	0.579	6.17	7.00	0.015	1.66	**1.30**	5.03	0.290	39.13	**8.03**	13.13	63.00
Q47	**0.003**	0.577	5.02	7.00	0.015	1.69	**1.32**	4.85	0.145	23.48	**7.90**	12.96	59.00
Q48	**0.004**	0.576	5.91	11.00	0.012	2.64	**1.94**	6.00	0.386	103.91	**14.95**	20.98	43.00
Q49	**0.004**	0.579	4.90	5.00	0.012	2.66	**1.95**	4.89	0.435	120.44	**16.98**	19.58	46.00
Q50	**0.003**	0.584	5.73	12.00	0.015	6.83	**5.38**	5.90	7.633	3505.65	450.28	**351.75**	1275.00
Q51	**0.003**	0.580	5.76	8.00	0.009	13.85	7.62	**5.18**	0.580	1125.65	100.27	**97.04**	161.00
Q52	**0.003**	0.579	5.76	15.00	0.009	16.47	9.34	**5.48**	0.145	261.74	97.02	**88.90**	124.00
Q53	**0.003**	0.574	6.84	13.00	0.012	10.09	6.55	**6.35**	0.193	206.52	61.78	**51.06**	92.00
Q54	**0.003**	*	451.39	1212.00	0.013	**79.73**	*	455.22	0.097	926.96	*	**570.96**	1396.00
Q55	**0.003**	*	411.78	1124.00	0.018	**0.20**	*	406.06	0.048	**2.61**	*	406.52	1130.00
Q56	**0.003**	*	409.39	1147.00	0.153	**6.53**	*	408.47	0.241	**11.74**	*	409.64	1144.00
Q57	**0.003**	*	416.89	1126.00	0.031	**5.39**	*	415.54	0.097	**33.48**	*	420.74	1126.00
Q58	**0.003**	*	426.56	1096.00	0.032	**30.76**	*	424.73	0.145	**165.65**	*	444.57	1200.00

⋆: Query with axes not supported by SXSI Fastest running times are marked in boldface

Table 9.8: Running times (in milliseconds) for the group of queries `C` over `XMark4` document.

	XMark4												
	Count				*Materialize*				*Materialize+Serialize*				
	XXS_{all}	SXSI	Monet	Qizx	XXS_{50}	XXS_{all}	SXSI	Monet	XXS_{50}	XXS_{all}	SXSI	Monet	Qizx
Q43	**0.001**	0.672	1479.27	19261.00	0.008	1985.51	**1511.98**	1535.915	15280.200	441356.00	**57049.30**	61894.76	218665.00
Q44	**0.003**	0.574	7.29	13.00	0.008	**1.40**	29.59	11.08	0.467	110.00	39.26	**27.73**	180.00
Q45	**0.003**	0.579	8.57	1.00	0.006	**0.006**	0.61	8.53	508.785	504.62	**386.04**	606.70	1264.00
Q46	**0.003**	0.571	12.17	11.00	0.016	14.77	**6.51**	11.02	0.280	363.08	**66.63**	1122.53	94.00
Q47	**0.003**	0.569	8.76	10.00	0.016	14.86	**6.61**	8.10	0.187	203.85	**64.23**	1191.19	93.00
Q48	**0.004**	0.572	13.14	17.00	0.012	24.02	12.29	**11.66**	0.280	939.23	**129.41**	1146.89	159.00
Q49	**0.004**	0.573	10.23	11.00	0.014	24.02	12.41	**10.04**	0.467	1086.15	**147.16**	1115.25	249.00
Q50	**0.003**	0.578	16.68	19.00	0.014	61.78	42.80	**16.26**	7.477	30645.30	**4008.04**	11671.49	19054.00
Q51	**0.003**	0.577	24.48	72.00	0.008	125.42	62.50	**25.11**	0.561	9969.23	**890.25**	11609.33	11181.00
Q52	**0.003**	0.577	27.96	96.00	0.012	147.94	78.06	**27.00**	0.093	2374.62	**870.13**	11453.98	8733.00
Q53	**0.003**	0.567	21.47	62.00	0.010	91.22	53.62	**19.15**	0.187	1849.23	**556.13**	3506.14	1641.00
Q54	**0.003**	*	5225.31	24360.00	0.014	**721.87**	*	5308.76	0.093	8348.46	*	**6284.36**	29676.00
Q55	**0.003**	*	4895.67	23720.00	0.018	**1.78**	*	4905.57	0.093	**21.54**	*	4786.40	23761.00
Q56	**0.003**	*	4910.53	23930.00	0.154	**58.97**	*	4862.37	0.280	**106.92**	*	4781.53	23720.00
Q57	**0.002**	*	4897.69	23762.00	0.031	**51.03**	*	4913.76	0.093	**303.85**	*	4854.26	28575.00
Q58	**0.003**	*	4988.83	24417.00	0.031	**286.08**	*	5015.04	0.187	**1488.46**	*	5125.83	24777.00

⋆: Query with axes not supported by SXSI Fastest running times are marked in boldface

Qizx/DB, and rewrote some of the queries of group `D`[41] to make them as efficient as possible, while preserving the semantics of the original ones. To this aim, we used the **`ftcontains`** text function instead of the standard **`contains`**, since it is more efficient[42]. For MonetDB/XQuery, the PF/Tijah text index [LMR+05] included

[41] Remember that this group is devoted to cover queries with **`contains`** and **`equal`** text functions.

[42] **`ftcontains`** allows to express *contains*-like queries, but also regular expression matching. It makes use of the full-text index.

also supports some full-text capabilities[43]. However it does not include an optimized version of the `contains` operator, hence we used the standard one, that relies on string conversions. Regarding SXSI, we must realize that its `contains` and `equal` implementations do not support text searches over phrases spanning more than one text node. Therefore, in case of SXSI times should not be considered in a strict sense, as text searches may potentially require less processing than that faced by the rest of the systems.

Table 9.9: Running times (in milliseconds) for the group of queries `D` over `XMark2` document.

	XMark2												
	Count				*Materialize*				*Materialize+Serialize*				
	XXS_{all}	SXSI	Monet	Qizx	XXS_{50}	XXS_{all}	SXSI	Monet	XXS_{50}	XXS_{all}	SXSI	Monet	Qizx
Q59	**29.91**	73.83	187.77	40.00	3.873	**29.91**	75.60	187.62	10.097	83.48	82.58	192.76	**62.00**
Q60	**6.50**	8.63	30.77	32.00	2.927	**6.50**	8.83	30.76	3.720	**7.39**	9.42	31.30	34.00
Q61	1.76	**1.50**	22.44	3.00	1.761	1.76	**1.58**	22.43	1.981	1.74	**1.64**	22.58	4.00
Q62	33.75	184.28	**17.62**	42.00	0.450	33.75	189.85	**17.35**	0.628	48.70	192.83	**20.48**	84.00
Q63	13.69	**2.27**	48.70	21.00	13.684	13.69	**2.37**	48.23	14.444	14.78	**2.50**	48.69	31.00
Q64	65.85	88.78	28.62	**8.00**	0.329	65.85	90.81	**28.88**	0.531	106.96	103.66	**38.80**	83.00
Q65	14.73	*	**12.81**	48.00	1.387	14.73	*	**12.87**	1.450	16.09	*	**13.24**	58.00
Q66	38.02	55.37	**18.66**	39.00	0.856	38.02	56.42	**18.54**	1.111	49.57	59.55	**20.70**	92.00
Q67	5.75	**2.21**	415.04	10.00	5.768	5.75	**2.30**	422.09	9.758	9.57	**2.74**	413.72	16.00
Q68	2.70	**1.59**	218.12	9.00	2.699	2.70	**1.68**	216.84	3.913	3.91	**1.67**	217.11	18.00
Q69	**22.89**	*	24.73	90.00	0.576	**22.89**	*	24.35	0.918	36.09	*	**25.06**	111.00
Q70	52.10	*	**23.62**	69.00	1.236	52.10	*	**23.99**	1.785	77.78	*	**27.88**	96.00
Q71	**3.42**	*	12.96	53.00	2.974	**3.42**	*	12.99	3.532	**4.07**	*	13.14	55.00
Q72	**3.10**	*	33.92	57.00	2.810	**3.10**	*	34.43	2.995	**3.48**	*	34.56	69.00
Q73	**3.98**	*	448.03	1305.00	0.573	**3.98**	*	453.72	0.773	**4.78**	*	453.91	1304.00

⋆: Query with axes not supported by SXSI   Fastest running times are marked in boldface

Table 9.9 and 9.10 presents the execution times obtained for each text oriented query. As it is shown, XXS performs on par with SXSI, and MonetDB/XQuery for tests over `XMark2` (as none of them actually stands out from the other), all of them outperforming Qizx/DB[44]. Yet in case of `XMark4` (the biggest file), MonetDB/XQuery obtains quite larger times, while both XXS and SXSI scale well. If we consider Qizx/DB, we can see that the bigger the document, the better the performance it exhibits. In particular, we must highlight its good time results for the count scenario of `XMark4`.

By analyzing each individual query in detail, we can observe that XXS performs better than the rest of the systems when evaluating queries involving elements content searches over single words, while phrase patterns may blur these time differences. This is mainly due to the fact that phrase processing requires several top-down traversals to verify the codewords around the occurrence of the least frequent word of the pattern. However, for attributes, XXS beats any of the tools.

[43] Such as a complex `about` operator for approximate matches, which ranks results by order of relevance.

[44] With the proviso of MonetDB/XQuery for some queries.

Table 9.10: Running times (in milliseconds) for the group of queries `D` over `XMark4` document.

	XMark4													
	Count				*Materialize*				*Materialize+Serialize*					
	XXS_{all}	SXSI	Monet	Qizx	XXS_{50}	XXS_{all}	SXSI	Monet	XXS_{50}	XXS_{all}	SXSI	Monet	Qizx
Q59	267.66	597.52	6481.68	**88.00**	3.947	**267.66**	610.25	6935.21	10.280	748.46	677.27	6981.74	**222.00**
Q60	44.77	72.61	4730.67	**36.00**	2.574	**44.77**	73.46	4735.92	3.271	**57.69**	78.92	4993.35	59.00
Q61	13.18	**7.95**	4598.22	20.00	3.613	13.18	**7.98**	4791.35	4.112	16.15	**8.34**	4791.75	27.00
Q62	304.39	1525.35	1398.49	**295.00**	0.396	**304.39**	1568.06	1399.03	0.561	**438.46**	1600.01	1504.30	481.00
Q63	120.37	**12.94**	2781.56	126.00	15.353	120.37	**13.07**	2859.08	16.168	126.15	**14.01**	2867.76	191.00
Q64	596.17	800.89	4631.92	**28.00**	0.329	**596.17**	817.25	4645.80	0.467	963.85	**930.57**	4738.66	1686.00
Q65	**129.72**	*	469.91	172.00	1.331	**129.72**	*	463.80	1.495	**140.77**	*	486.67	204.00
Q66	337.20	499.99	2127.11	**143.00**	0.838	**337.20**	506.31	2207.13	1.028	450.00	535.68	2229.77	**246.00**
Q67	53.27	**11.96**	12148.06	71.00	8.463	53.27	**12.04**	12220.45	15.794	95.39	**15.91**	12322.78	119.00
Q68	18.41	**7.95**	8950.99	19.00	18.491	18.41	**7.93**	8970.11	26.822	26.92	**7.94**	8971.28	42.00
Q69	**205.23**	*	371.11	352.00	0.671	**205.23**	*	375.70	0.927	**323.08**	*	382.55	459.00
Q70	**466.73**	*	1163.32	4279.00	1.251	**466.73**	*	1211.98	1.782	**699.23**	*	4858.95	4286.00
Q71	**2.80**	*	775.83	246.00	2.768	**2.80**	*	776.26	3.783	**3.85**	*	778.77	262.00
Q72	**2.15**	*	250.33	291.00	2.221	**2.15**	*	252.89	2.430	**2.31**	*	253.04	292.00
Q73	**2.80**	*	5186.66	25522.00	0.763	**2.80**	*	5221.96	0.934	**3.85**	*	5298.02	25936.00

⋆: Query with axes not supported by SXSI   Fastest running times are marked in boldface

Unlike the groups of structural based queries, textual queries are commonly much more selective in terms of number of results produced. Hence, XXS materialization plus serialization times are not as affected by the times required to recompose the codeword bytes of the words before decoding them, as happened there. Anyway, note that we also show the XXS running times of retrieving the first 50 results for each query, requiring just a few milliseconds.

Chapter 10

Conclusions and Future Work

10.1 Summary of Contributions

The use of the *eXtensible Markup Language* (XML) has been constantly growing in the last years, due to its great flexibility for semi-structured data representation and its acknowledged suitability for data exchange on the Internet. As its relevance increased, query languages were also proposed to process and extract relevant information from this kind of documents, as well as solutions to give them support. Some of these approaches focused on the query aspect, and devoted their efforts to provide efficient query evaluation solutions, without regarding space requirements. In turn, other works pursued the same aim, but also considered one of the main drawbacks of XML, its *verbosity*, and tried to use the minimum amount of space as possible, in the form of compressed representations. The main advantage of these tools arises from space reductions, but they also add some extra benefits: dropping the space may be key to fit data structures in main memory rather than swapping out to disk, operating in higher and faster levels of the memory hierarchy; to use fewer machines, or even to achieve a feasible solution when the memory is limited (as in mobile devices). Hence, the relevance of these approaches has triggered a large amount of research in this area. However, today there is no available solutions providing efficient query support within the space of the compressed text.

In this thesis we address the problem of a stated lack of practical tools with the aforementioned features and present what can be considered the first practical available solution for compressed self-indexed storage of XML documents, which takes a very little amount of space, and which provides, at the same time, efficient query support, by specially focusing on XPath evaluation. Our system, which we called XXS, includes two main contributions to the state of the art:

- First, we have proposed the XML Wavelet Tree (XWT), a new compressed self-indexed representation of XML documents, which permits compact storage, providing in addition efficient querying capabilities. Our structure occupies a space proportional to the compressed text, (about 30%-40% of the original document size), keeping almost the same compression ratios as other word-based byte-oriented semistatic statistical compression methods (just requiring about 4%-5% more, on average), and taking reasonable times to compress the document (since a more complex parsing of the input document is performed to meet XML features), and better decompression times.

  In comparison with other general text compressors and XML conscious but non-queriable compressors, XWT compression ratios are not as good as that obtained by these other tools, as might be expected, since none of them exhibits the querying ability. They rather aim to compress to the best. However, XWT drastically improves the compression and decompression times of virtually all of them.

  Yet the most important feature of the XWT representation is that with just a little amount of extra space (about 4%-8%) to provide this structure with powerful indexing capabilities[1] (for the structures of partial counters used to speed up basic operations, and for the succinct tree representation of the document structure), it is able to further allow efficient XML querying purposes.

- Second, we have designed and implemented a complete *Query module* for the efficient evaluation of XPath queries over a XWT representation, taking advantage of its valuable self-indexing properties. This module has been divided into two main parts, namely the *Query parser* and the *Query evaluator*, whose detailed descriptions has been presented from Chapter 6 to Chapter 8. For the *Query parser* submodule, in charge of the query parsing tasks, the process from the preliminary representation of a query (the *query parse tree*) up to the obtention of the final query execution plan (the *query execution tree*) has been explained. For the *Query evaluator* submodule, devoted to perform the actual evaluation tasks, we have described the performance of the global evaluation procedure, characterized by three main strategies: a *bottom-up* approach, a *lazy evaluation* scheme, and a *skipping* strategy; and also we have fully provided the implementations of every operator, with comprehensive discussions.

As a whole, XXS provides efficient XPath evaluation within the space of the compressed document. Experiments show that our system successfully competes with some well known solutions in the state of the art supporting XPath, and that it largely outperforms them in terms of amount of space used.

[1] We also include the space needed to maintain the vocabularies into hash tables.

In particular, experimental results have shown XXS outstanding performance. If we consider the retrieval of the whole set of query results, it has been proved that, most times, XXS performs better than the best current alternatives, for both counting and materializing scenarios. Only when serialization is involved, XXS does not exhibit such a remarkable behavior (although its results are still competitive), since unlike other systems, which have a straightforward access to the text, XXS needs to recompose the codeword bytes spread along the different XWT nodes before decoding and displaying a word. But in this case, we must stand out one of the main XXS features that makes these time differences be actually blurred: the ability of obtaining results on user demand, thanks to its *lazy evaluation* scheme. As experiments showed, XXS is able to immediately report (within one millisecond in most queries) a first batch of query results, and to continue producing the rest while the others are still being consumed by the user.

The other striking characteristic of XXS is that it is by far the system that uses less amount of space (and also less time to be constructed). Note that the compressed (and self-indexed) storage of XXS arises from the XWT data structure (plus the above mentioned additional waste of extra space used to improve its efficiency). Experiments showed that the rest of the systems (including XML conscious queriable compressors, such as SXSI) require between 2 and 5 times more space than that used by XXS. Hence, it results in a good alternative to work with huge corpus that otherwise should be manipulated on disk.

Summarizing, XXS requires little space, provides efficient and outstanding XPath querying capabilities, and shows a robust and scalable behavior, features that lead XXS to have no current competitors with comparable query evaluation performance in the same amount of space.

10.2 Future Work

In this section we detail some of the plans considered for future work after this thesis:

- Given the good performance of XXS, we plan to extend the practical subset of XPath targeted in this work to also meet some of the XPath extensions, such as inequalities and positional predicates, and to consolidate its status, even more, giving support to the XQuery language. As XPath constitutes the core of XQuery, we intend to apply the efficient querying capabilities of XXS to solve FLWOR clauses.

- Another quite interesting future plan is to introduce document retrieval into our structures. In this way, query evaluation could also provide document information, more suitable for some scenarios. For instance, if we want to find relevant documents to user queries when working with collections of several documents. In addition, the introduction of relevance measures is also planned, to provide each retrieved result (in the form of specific XML components or documents) with a retrieval status value, and even to carry out ranking tasks with respect to a query.

An initial approach to this goal has already been studied, and presented in [BCPNP12].

- As a way to promote the use of XXS by the community, we aim to create both an API interface, to allow its integration with other systems such as digital libraries; and also a complete application focused on providing compressed storage of XML documents and XPath querying facilities, based on the work developed.

Appendix A

Publications and Other Research Results

Publications

Journals to be submitted

- Brisaboa, N. R., Cerdeira-Pena, A., Navarro, G. XXS: Efficient XPath Evaluation over Compressed Self-Indexed XML documents. Manuscript to be submitted to a high-level journal.
- Brisaboa, N. R., Cerdeira-Pena, A., Navarro, G., Pedreira, O. Space Efficient Ranked Document Retrieval. Manuscript to be submitted to a high-level journal.

International conferences

- Brisaboa, N. R., Cerdeira-Pena, A., Navarro, G. A Compressed Self-indexed Representation of XML Documents. In *Proc. of the 13th European Conference on Digital Libraries (ECDL)* - LNCS 5714, pp. 273-284, Corfu, Greece, 2009.
- Brisaboa, N. R., Cerdeira-Pena, A., Navarro, G., Pasi, G. An Efficient Implementation of a Flexible XPath Extension. In *Proc. of the 9th International Conference on Adaptivity, Personalization and Fusion of Heterogeneous Information (RIAO)*, pp. 140-147, Paris, France, 2010.
- Brisaboa, N. R., Cerdeira-Pena, A., Navarro, G., Pedreira, O. Ranked Document Retrieval in (Almost) No Space. In *Proc. of the 19th International*

Symposium on String Processing and Information Retrieval (SPIRE) - LNCS 7608, pp. 155-160. Cartagena de Indias, Colombia, 2012.

- Brisaboa, N. R., Cerdeira-Pena, A., Navarro, G. XPath Evaluation over Compressed and Self-indexed XML documents. In 7^{th} *Workshop on Compression, Text, and Algorithms of the* 19^{th} *International Symposium on String Processing and Information Retrieval (SPIRE)*. Cartagena de Indias, Colombia, 2012.

National conferences

- Álvarez, S., Cerdeira-Pena, A., Fariña, A., Ladra, S. Desarrollo de un Compressor de Textos Orientado a Palabras basado en PPM. In *Actas de las XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pp. 237-248. San Sebastián, Spain, 2009.
- Brisaboa, N. R., Cerdeira-Pena, A., Navarro, G., Pasi, G. Estrategias de Optimización de Consultas XPath Flexibles sobre XML Wavelet Trees. In *Actas del I Congreso Español de Recuperación de Información (CERI)*, pp. 207-218. Madrid, Spain, 2010.
- Brisaboa, N. R., Cerdeira-Pena, A., Navarro, G. A Compressed Self-indexed Representation of XML Documents. In *Actas de las XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pp. 199-199. Valencia, Spain, 2010.

Research stays

- *February, 2008 - July, 2008.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).
- *September, 2009 - December, 2009.* Research stay at Università degli Studi di Milano, Information Retrieval Group (Milano, Italy).
- *March, 2010 - May, 2010.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).
- *January, 2011.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).
- *January, 2012.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).

Appendix B

Algorithms

This appendix details the pseudocode of some operators whose implementation is discussed in Chapter 8.

Algorithm B.1: *Next* procedure of *any* element (i.e. '*' applied to elements)

Input: new_s, new_e (new positional restrictions), $last_s$ (start position of the last delivered segment), $stack$

Output: next valid occurrence of *any* element

```
if new_s ≥ new_e then
    inspectStack(new_s)
    occ_s ← rank_( (new_s)
    if occ_s + 1 ≤ n_( then
        pos_s ← select_( (occ_s + 1)
        pos_e ← findclose(pos_s)
        result ← segment(pos_s, pos_e)
    else
        result ← ∅
    end
else
    max_s = max(new_s − 1, last_s)
    inspectStack(new_e, new_s)
    occ_e ← rank_) (new_e)
    if occ_e + 1 ≤ n_) then
        pos_e ← select_) (occ_e + 1)
        pos_s ← findopen(pos_e)
```

18.   **if** $pos_s \leq max_s$ **then**
19. $new_s \leftarrow pos_e + 1$; *go to* 3.
20.   **end**
21.   **else**
22. $result \leftarrow \emptyset$
23.   **return** $result$
24.   **end**
25. $occ_s \leftarrow rank_{(}\ (pos_e)$
26. $occ_{nested} \leftarrow occ_s - occ_e - 1$
27. $match \leftarrow pos_s; i \leftarrow 0$
28.   **while** $i < occ_{nested}$ **do**
29. $parent_s \leftarrow enclose(match)$
30.   **if** $parent_s > max_s$ **then**
31. $stack.push(segment(pos_s, pos_e))$
32. $pos_s \leftarrow parent_s$
33. $pos_e \leftarrow findclose(pos_s)$
34. $match \leftarrow parent_s$; $i \leftarrow i + 1$
35.   **else**
36.   **break**
37.   **end**
38.   **end**
39. $result \leftarrow segment(pos_s, pos_e)$
40. **end**
41. **return** $result$

Algorithm B.2: *Next* procedure of `or` operator (*non-nested* variant)

```
Input: new_s, new_e (new positional restrictions)
Output: next element segment from any side
left ← L.result
right ← R.result
if lastL or (left ≠ø and (left.s < new_s or left.e < new_e)) then
    left_s ← max(left_s, new_s)
    left_e ← max(left_e, new_e)
    left ← L.next(left_s, left_e)
if lastR or (right ≠ø and (right.s < new_s or right.e < new_e)) then
    right_s ← max(right_s, new_s)
    right_e ← max(right_e, new_e)
    right ← R.next(right_s, right_e)
lastL ← 0; lastR ← 0
if left ≠ø and right ≠ø then
    case left < right or left ⊃ right
        left_s ← left.e + 1; lastL ← 1; result ← left; return result
    case left > right or left ⊂ right
        right_s ← right.e + 1; lastR ← 1; result ← right; return result
else
    if left ≠ø then
        left_s ← left.e + 1; lastL ← 1; result ← left; return result
    else
        if right ≠ø then
            right_s ← right.e + 1; lastR ← 1; result ← right; return result
        else
            result ←ø; return result
```

Algorithm B.3: *Next* procedure of $\texttt{or}_{\texttt{att}}$ operator

Input: $news$ (new positional restriction)
Output: next attribute segment from any side

$left \leftarrow L.result$
$right \leftarrow R.result$
if $lastL$ **or** $(left \neq \emptyset$ **and** $left.s_{root} < new_s)$ **then**
  $left_s \leftarrow max(left_s, new_s)$
  $left \leftarrow L.next(left_s)$
if $lastR$ **or** $(right \neq \emptyset$ **and** $right.s_{root} < new_s)$ **then**
  $right_s \leftarrow max(right_s, new_s)$
  $right \leftarrow R.next(right_s)$
$lastL \leftarrow 0$; $lastR \leftarrow 0$
if $left \neq \emptyset$ **and** $right \neq \emptyset$ **then**
  case $left < right$
    $left_s \leftarrow left.s_{root} + 1$; $lastL \leftarrow 1$; $result \leftarrow left$; **return** $result$
  case $left > right$
    $right_s \leftarrow right.s_{root} + 1$; $lastR \leftarrow 1$; $result \leftarrow right$; **return** $result$
else
  if $left \neq \emptyset$ **then**
    $left_s \leftarrow left.s_{root} + 1$; $lastL \leftarrow 1$; $result \leftarrow left$; **return** $result$
  else
    if $right \neq \emptyset$ **then**
      $right_s \leftarrow right.s_{root} + 1$; $lastR \leftarrow 1$; $result \leftarrow right$; **return** $result$
    else
      $result \leftarrow \emptyset$; **return** $result$

Algorithm B.4: *Next* procedure of $\texttt{or}_{\texttt{phrase}}$ operator

Input: new_s (new positional restriction)
Output: next phrase segment from any side

```
left ← L.result
right ← R.result
if lastL or (left ≠ø and left.s_root < new_s) then
    left_s ← max(left_s, new_s)
    left ← L.next(left_s)
if lastR or (right ≠ø and right.s_root < new_s) then
    right_s ← max(right_s, new_s)
    right ← R.next(right_s)
lastL ← 0; lastR ← 0
if left ≠ø and right ≠ø then
    case left < right or left ⊃ right
        left_s ← left.e_root + 1; lastL ← 1; result ← left; return result
    case left > right or left ⊂ right
        right_s ← right.e_root + 1; lastR ← 1; result ← right; return result
else
    if left ≠ø then
        left_s ← left.e_root + 1; lastL ← 1; result ← left; return result
    else
        if right ≠ø then
            right_s ← right.e_root + 1; lastR ← 1; result ← right; return result
        else
            result ←ø; return result
```

Algorithm B.5: *Next* procedure of `contains` text function for single words (*non-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `contains` semantics
// We assume that b_y is the first byte of a start/end-tag's codeword

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_e \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
    $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
    if $right \neq \emptyset$ **then**
      $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$
  case $left \supseteq right$ // left.s<=right.s **and** left.e>right.e
    $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
  otherwise
    $left_s \leftarrow left.e + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return $result$

Algorithm B.6: *Next* procedure of `contains` text function for a phrase (*non-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `contains` semantics
// We assume that b_y is the first byte of a start/end-tag's codeword

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_e \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
    $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
    if $right \neq \emptyset$ **then**
      $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$;
      $right.e \leftarrow rank_{b_y}(XWTroot, right.e_{root})$
  case $left \supseteq right$ // left.s<=right.s **and** left.e>right.e
    $left_e \leftarrow left.e + 1$; $result \leftarrow left$; **return** *result*
  case $left.s > right.s$ **and** $left.e > right.e$
    $right_s \leftarrow right.e_{root} + 1$; $right \leftarrow R.next(right_s)$
    if $right \neq \emptyset$ **then**
      $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$;
      $right.e \leftarrow rank_{b_y}(XWTroot, right.e_{root})$
  otherwise
    $left_s \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return *result*

Algorithm B.7: *Next* procedure of `equal` text functions for single words (*non-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `equal` semantics
// We assume that b_y is the first byte of a start/end-tag's codeword

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_e \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
    $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
    if $right \neq \emptyset$ **then**
      $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$
  case $left \supseteq right$ // left.s<=right.s **and** left.e>right.e
    $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
    $left.e_{root} \leftarrow select_{b_y}(XWTroot, left.e)$
    // *checkBoundaries* : verifies equality condition, despite of interleaved
    // occurrences of start/end-tags, comments and processing instructions,
    // which are `skipped`.
    if $checkBoundaries(left.s_{root}, left.e_{root}, right.s_{root}, right.e_{root})$ **then**
      $left_s \leftarrow left.e + 1$; $result \leftarrow left$; **return** $result$
    else
      $right_s \leftarrow left.e_{root} + 1$; $right \leftarrow R.next(right_s)$
      if $right \neq \emptyset$ **then**
        $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$
  otherwise
    $left_s \leftarrow left.e + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return $result$

Algorithm B.8: *Next* procedure of `equal` text function for single words (*full-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `equal` semantics
// We assume that b_y is the first byte of a start/end-tag's codeword

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
    case $left < right$
        $left_e \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
    case $left > right$
        $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
        $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
        if $right \neq \emptyset$ **then**
            $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$
    case $left \supseteq right$ // left.s<=right.s **and** left.e>right.e
        $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
        $left.e_{root} \leftarrow select_{b_y}(XWTroot, left.e)$
        // *checkBoundaries* : verifies equality condition, despite of interleaved
        // occurrences of start/end-tags, comments and processing instructions,
        // which are `skipped`.
        if $checkBoundaries(left.s_{root}, left.e_{root}, right.s_{root}, right.e_{root})$ **then**
            $left_e \leftarrow right.s + 1$; $result \leftarrow left$; **return** $result$
        else
            $left_e \leftarrow right.s + 1$; $left \leftarrow L.next(left_s, left_e)$
    otherwise
        $left_s \leftarrow left.e + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return $result$

Algorithm B.9: *Next* procedure of `equal` text function for a phrase (*non-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `equal` semantics
// We assume that b_y is the first byte of a start/end-tag's codeword

```
left_s ← max(left_s, new_s)
left_e ← max(left_e, new_e)
left ← L.next(left_s, left_e)
right ← R.result
while left ≠ø and right ≠ø do
    case left < right
        left_e ← right.e + 1; left ← L.next(left_s, left_e)
    case left > right
        left.s_root ← select_{b_y}(XWTroot, left.s)
        right_s ← left.s_root + 1; right ← R.next(right_s)
        if right ≠ø then
            right.s ← rank_{b_y}(XWTroot, right.s_root);
            right.e ← rank_{b_y}(XWTroot, right.e_root)
    case left ⊇ right // left.s<=right.s and left.e>right.e
        left.s_root ← select_{b_y}(XWTroot, left.s)
        left.e_root ← select_{b_y}(XWTroot, left.e)
        // checkBoundaries : verifies equality condition, despite of interleaved
        // occurrences of start/end-tags, comments and processing instructions,
        // which are skipped.
        if checkBoundaries(left.s_root, left.e_root, right.s_root, right.e_root) then
            left_s ← left.e + 1; result ← left; return result
        else
            right_s ← left.e_root + 1; right ← R.next(right_s)
            if right ≠ø then
                right.s ← rank_{b_y}(XWTroot, right.s_root);
                right.e ← rank_{b_y}(XWTroot, right.e_root)
    otherwise
        left_s ← right.e + 1; left ← L.next(left_s, left_e)
result ← ø
return result
```

Algorithm B.10: *Next* procedure of `equal` text function for a phrase (*full-nested* variant)

Input: new_s,new_e (new positional restrictions)
Output: next occurrence of the left side fulfilling `equal` semantics
// We assume that b_y is the first byte of a start/end-tag's codeword

$left_s \leftarrow max(left_s, new_s)$
$left_e \leftarrow max(left_e, new_e)$
$left \leftarrow L.next(left_s, left_e)$
$right \leftarrow R.result$
while $left \neq \emptyset$ **and** $right \neq \emptyset$ **do**
  case $left < right$
    $left_e \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
  case $left > right$
    $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
    $right_s \leftarrow left.s_{root} + 1$; $right \leftarrow R.next(right_s)$
    if $right \neq \emptyset$ **then**
      $right.s \leftarrow rank_{b_y}(XWTroot, right.s_{root})$;
      $right.e \leftarrow rank_{b_y}(XWTroot, right.e_{root})$
  case $left \supseteq right$ // left.s<=right.s **and** left.e>right.e
    $left.s_{root} \leftarrow select_{b_y}(XWTroot, left.s)$
    $left.e_{root} \leftarrow select_{b_y}(XWTroot, left.e)$
    // *checkBoundaries* : verifies equality condition, despite of interleaved
    // occurrences of start/end-tags, comments and processing instructions,
    // which are `skipped`.
    if $checkBoundaries(left.s_{root}, left.e_{root}, right.s_{root}, right.e_{root})$ **then**
      $left_e \leftarrow right.e + 1$; $result \leftarrow left$; **return** $result$
    else
      $left_e \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
  otherwise
    $left_s \leftarrow right.e + 1$; $left \leftarrow L.next(left_s, left_e)$
$result \leftarrow \emptyset$
return $result$

Appendix C

Descripción del Trabajo Realizado

C.1 Introducción

Desde su aparición en 1998, la importancia del lenguaje de marcado *eXtensible Markup Language* (XML) [XMLa] ha ido creciendo de manera constante gracias a las enormes posibilidades que ofrece para el intercambio de datos en Internet y, en general, para la comunicación de información semi-estructurada entre aplicaciones de diferentes plataformas. De hecho, hoy día se considera el estándar *de facto* para la representación de datos semi-estructurados, siendo utilizado para el almacenamiento de grandes volúmenes de información en dominios que abarcan desde el comercio electrónico, las bibliotecas digitales, o los catálogos, hasta aplicaciones biológicas y médicas, especificaciones de metadatos, etc.

Una de las principales características del XML es su expresividad. Para poder aprovechar ésta al máximo, son varios los lenguajes de consulta que se han venido definiendo a lo largo de los años. Es el caso de dos de sus máximos exponentes, XPath y XQuery, lenguajes que permiten la realización de consultas tanto sobre el contenido del documento como sobre su estructura. La importancia creciente de estos lenguajes unido al reto que supone dar un soporte eficiente a ellos, ha motivado numerosos trabajos de investigación con el objeto de proporcionar soluciones competitivas, bien como propuestas teóricas, bien en forma de sistemas reales. Estos sistemas se dividen generalmente en dos grandes categorías: aquéllos que siguen una aproximación en *streaming* (por ejemplo, GCX [SSK07], SPEX [SPE], etc.), y que por tanto deben realizar una lectura secuencial del documento previa respuesta de una consulta; y los *indexados* (tales como Saxon [Kay08], Galax [FSC+03], MonetDB/XQuery [BGvK+06], Qizx/DB [Qiz], etc.), los cuales llevan a cabo un procesamiento previo del documento con el fin de crear estructuras de datos adicionales, que después son utilizadas para poder responder las consultas sin necesidad de recorrer secuencialmente los documentos de cada vez.

Los sistemas indexados resultan atractivos en multitud de escenarios, especialmente en aquellos casos en los que el coste de un recorrido secuencial puede llegar a ser prohibitivo o, incluso, cuando son muchas las consultas que se van a formular sobre un mismo documento. Sin embargo, y aunque en primera instancia los sistemas secuenciales puedan ser considerados como soluciones más lentas respecto de las soluciones indexadas, no siempre tiene por qué darse esta situación. Es importante mencionar que las soluciones indexadas mejoran las capacidades de consulta a expensas de incrementar sus necesidades de espacio, debido al uso de índices. Así pues, en caso de que el tamaño de estos obligue a su manipulación en disco, la eficiencia puede verse directamente afectada por tiempos de transferencia de E/S, incurriendo en tiempos incluso mayores que las aproximaciones secuenciales. Debido a ello, muchas investigaciones han dirigido sus esfuerzos a conseguir minimizar los elevados requisitos de espacio de los sistemas indexados, mediante la creación de índices en memoria principal.

En relación con el consumo de espacio, otra línea de investigación destacada ha sido el desarrollo de métodos de compresión para XML. Una de las propiedades fundamentales del modelo de datos XML es su gran flexibilidad. No obstante, ésta característica también constituye uno de sus principales inconvenientes, ya que puede dar lugar a documentos de gran tamaño que es preciso almacenar, transmitir, y consultar. En este sentido, el uso de herramientas de compresión no sólo supone un ahorro de espacio, sino también de tiempo, ya que procesar una versión comprimida de un documento permite reducir tiempos de transmisión, de acceso a disco, o más importante aún, de procesamiento. Así, son varios los trabajos que se han venido desarrollando a lo largo de estos últimos años en este ámbito. Desde la aplicación de compresores de texto generales, categorizados como compresores *XML ciegos* (por ejemplo, las técnicas Ziv-Lempel [ZL77, ZL78, Wel84], los códigos Huffman [Huf52, dMNZBY00], los métodos basados en PPM [CW84], o la familia de los Dense Codes [BFNP07]), hasta la creación de herramientas específicas para explotar las características propias de esta clase de documentos (también conocidos como compresores *XML específicos*). Asimismo, y dentro de estas últimas, algunos de los compresores dedicados han tratado de ir un paso más allá, y proporcionar a mayores un soporte para la realización de consultas (es el caso de compresores como XGrind [TH02], XPRESS [MPC03], XCQ [LNWL03, NLWL06], XQzip [CN04], XQueC[ABMP07], etc.). Algunos de ellos permiten que las consultas puedan ser resueltas directamente sobre la representación comprimida del documento (bien de manera secuencial, bien a través de índices). Otros, en cambio, precisan realizar algún tipo de descompresión (bien total, bien parcial) antes de poder operar sobre él. Sin embargo, y a pesar de las numerosas propuestas, se ha constatado que, en la actualidad, existe una carencia importante de herramientas prácticas a disposición de los usuarios [Sak09], especialmente en el caso de mayor relevancia, el de técnicas de compresión específicas para XML, con capacidades de consulta.

Otra línea de trabajo más reciente ha sido la de combinar compresión e indexación, a través de lo que se conoce como *autoíndices*. En este caso, hablamos de estructuras que ocupan un espacio proporcional al texto comprimido, lo reemplazan y además proporcionan un acceso eficiente al mismo [NM07]. Uno de los objetivos principales de estos índices es su almacenamiento en memoria principal, para evitar los elevados costes de acceso a disco. En este dominio, todavía son pocos los trabajos desarrollados en torno al ámbito de tratamiento de documentos XML. Un ejemplo lo encontramos en la herramienta XBzipIndex [FLMM05, FLMM06], un autoíndice específico para documentos XML, que además permite la realización de consultas, si bien limitadas a tipos muy específicos. Otra propuesta reciente, enfocada en la indexación comprimida de documentos XML, ha sido la herramienta SXSI [ACM$^+$10]. Diseñada para trabajar en memoria principal, esta herramienta es capaz de responder un subconjunto más amplio de consultas. No obstante, en este caso, su principal inconveniente reside en los elevados requisitos de espacio que presenta, si se compara con el tamaño del texto comprimido.

Así pues, se puede observar que a día de hoy todavía existe una clara necesidad de implementaciones eficientes, escalables y estables que ocupen poco espacio y además ofrezcan, al mismo tiempo, un soporte competitivo para la consulta de documentos XML.

C.2 Metodología

En este trabajo se ha tenido en cuenta la casuística presentada en la Sección C.1, y en concreto, la ausencia de herramientas prácticas disponibles que aúnen a la vez importantes capacidades de consulta, junto con unos requisitos de espacio mínimos. De esta manera, se ha desarrollado lo que puede considerarse como la primera solución práctica disponible para el almacenamiento comprimido y auto-indexado de documentos XML, capaz de ofrecer un soporte eficiente a la evaluación de consultas XPath en el espacio del texto comprimido (alrededor de un 30%-40% del tamaño original del documento). El planteamiento seguido para su consecución se muestra a continuación:

- Inicialmente, se realizó un completo estudio bibliográfico en relación a las propiedades y modelo de datos del lenguaje de marcado XML, así como de los lenguajes definidos para su tratamiento y consulta, con especial énfasis en el lenguaje de consulta XPath. El objetivo era adquirir un profundo conocimiento acerca de las características del XML para la representación de información semi-estructurada y las necesidades básicas del procesamiento de documentos en este formato.

- Tras este primer paso, se procedió a una revisión exhaustiva de trabajos existentes en el ámbito del almacenamiento y consulta de documentos XML. Desde aproximaciones enfocadas a proporcionar un soporte eficiente a la consulta de este tipo de documentos, pero sin abordar con igual énfasis la problemática del espacio de almacenamiento, hasta aquellas otras soluciones cuya motivación principal reside precisamente en este último aspecto, pudiendo permitir algunas de ellas la realización de consultas. Se pretendía de esta manera establecer las bases del estado del arte, para poder contrastar las ventajas y debilidades de las distintas alternativas, así como determinar las necesidades principales a las que se debía dar solución. Dicho análisis permitió identificar la ausencia constatada de herramientas disponibles que ofreciesen un soporte de consultas eficiente, empleando requisitos de espacio mínimos.
- Los análisis realizados nos llevaron a proponer una nueva solución para el almacenamiento, procesamiento y consulta de documentos XML, eficiente en tiempo y en espacio, centrándonos en particular, en el lenguaje de consulta XPath. Así pues, la primera contribución de nuestro trabajo consistió en una nueva propuesta para la representación comprimida y auto-indexada de documentos XML, denominada XML Wavelet Tree (XWT). Esta estructura no sólo ofrece un almacenamiento compacto (obtiene ratios de compresión del 30%-40%), sino que además proporciona al mismo tiempo importantes capacidades de consulta. La constatación de este hecho se pone de manifiesto a través de la segunda de las contribuciones, el diseño e implementación de un módulo de consulta para la eficiente evaluación de consultas XPath sobre una representación XWT. En conjunto, ambos trabajos desarrollados constituyen un sistema completo, XXS (***X**Path evaluation on **X**ML documents using a **S**elf-index*), capaz de resolver eficientemente consultas del lenguaje XPath sobre documentos XML comprimidos y auto-indexados.
- Ya por último, se llevó a cabo la validación del sistema propuesto a través de una batería exhaustiva de experimentos. Los resultados obtenidos demostraron que nuestra solución es capaz de competir exitosamente con algunas de las herramientas más conocidas del estado del arte con soporte a la realización de consultas XPath, superándolas además ampliamente en términos de espacio. Este hecho evidencia, por tanto, la consecución de los objetivos iniciales, logrando cubrir la necesidad actual de una herramienta con estas características.

C.3 Conclusiones y Contribuciones

Conforme el uso del *eXtensible Markup Language* se ha venido haciendo más popular como estándar para la representación de datos semi-estructurados y el intercambio de datos en Internet, también lo han hecho los lenguajes de consulta propuestos para su explotación y procesamiento, así como los trabajos desarrollados con el fin de proporcionar soluciones a los retos que su tratamiento impone. Algunas de estas aproximaciones se han centrado en el desarrollo de propuestas enfocadas hacia una eficiente evaluación de consultas, como objetivo principal, relegando a un segundo plano las necesidades de espacio. Otras, sin embargo, han dirigido sus esfuerzos a tratar de abordar uno de los principales inconvenientes del lenguaje XML, su *verbosidad*, y usar el mínimo espacio posible a través de representaciones comprimidas, proporcionando en la medida de lo posible, posibilidades de consulta. En este sentido, la principal ventaja de estas últimas no sólo parte de una evidente reducción de espacio, sino también de una serie de beneficios adicionales. Por ejemplo, la reducción del espacio ocupado es fundamental para permitir el almacenamiento de estructuras de datos en niveles superiores (y, por tanto, más rápidos) de la jerarquía de memoria (como memoria principal), y evitar los costosos accesos a disco. A mayores, puede permitir también el uso de un menor número de máquinas en determinados contextos, o incluso ser un aspecto crucial para lograr una solución factible cuando el uso de memoria es limitado (como sucede en el caso de dispositivos móviles). Esto ha hecho que las aportaciones en este área a lo largo de los años hayan sido numerosas. No obstante, y a pesar de ello, hoy día no existe una solución disponible competitiva en términos de capacidad y tiempo de resolución de consultas, cuyas necesidades de espacio sean mínimas, cercanas al tamaño del texto comprimido.

En este trabajo de investigación hemos tratado de dar solución a esta problemática. Como resultado, esta tesis presenta una nueva propuesta para el almacenamiento comprimido y auto-indexado de documentos XML con soporte a la evaluación eficiente de consultas, en concreto, del lenguaje XPath. Nuestra solución, que hemos dado en llamar XXS, se compone de dos contribuciones principales:

- Primero, se ha propuesto el XML Wavelet Tree (XWT), una nueva representación comprimida y auto-indexada para documentos XML, que proporciona al mismo tiempo almacenamiento compacto, y características implícitas de auto-indexación. Nuestra estructura, basada en el uso de una técnica de compresión orientada a byte y a palabra, conocida como (s,c)-Dense Code, modificada específicamente para dotar al XWT de capacidades de consulta, permite la representación de documentos XML ocupando un 30%-40% de su tamaño original. Esto supone un incremento despreciable (alrededor del 4%-5%) con respecto a los ratios de compresión obtenidos por el propio (s,c)-Dense Code, y otras métodos de compresión de texto genéricos de las mismas características. Mientras, nuestra propuesta mantiene unos tiempos de compresión razonables (algo mayores debido a la complejidad que conlleva el procesamiento de documentos XML contemplando la identificación de los distintos componentes del modelo de datos subyacente), y mejora los tiempos de descompresión.

En comparación con otros compresores de propósito general y herramientas de compresión específica XML, aunque no capaces de soportar ningún tipo de consulta, XWT presenta, generalmente, unos ratios de compresión mayores, como es de esperar; ya que el propósito de los demás compresores es precisamente obtener la máxima compresión, sin proporcionar capacidades de consulta. No obstante, en relación a los tiempos, XWT mejora drásticamente tanto los tiempos de compresión como los de descompresión de prácticamente todos ellos.

En cualquier caso, la característica más importante de XWT es que con una mínima cantidad de espacio adicional (en torno al 4%-8%), necesaria para conferir a nuestra estructura destacadas capacidades de indexación[1] (mediante la creación de estructuras de contadores utilizadas para agilizar operaciones básicas de *rank* y *select*, así como de aquéllas empleadas para la representación sucinta de la propia estructura del documento XML), puede ser usado eficientemente con propósitos de evaluación de consultas; tal y como se pone de manifiesto a través de la segunda de las contribuciones.

- Como segunda contribución, se ha diseñado y desarrollado un módulo para la eficiente evaluación de consultas XPath sobre documentos comprimidos con XWT. En este sentido, hemos presentado una completa descripción de dicho módulo, distinguiendo sus dos principales componentes: el submódulo encargado del ánalisis sintáctico de la consulta, y su transformación en un plan de ejecución; y el submódulo dedicado a la propia evaluación de la consulta. En esta última parte, además de describir el procedimiento de evaluación general, caracterizado principalmente por tres estrategias: una aproximación *bottom-up*, un esquema de evaluación *lazy* y una estrategia de salto que permite procesar únicamente aquellas partes del documento relevantes para la consulta; se ha proporcionado también una detallada explicación de la implementación de cada posible operación[2].

En su conjunto, el sistema propuesto, XXS, proporciona un soporte a la evaluación de consultas XPath eficiente, en el espacio del texto comprimido. Los experimentos realizados prueban que XXS compite favorablemente con algunas de las soluciones más destacadas en el estado del arte para la resolución de consultas XPath, empleando además una cantidad de espacio mucho menor.

En concreto, los resultados experimentales evidencian el comportamiento realmente bueno de nuestro sistema. Si consideramos la situación en que la totalidad de resultados de una consulta es recuperado en su conjunto, se ha constatado que XXS logra mejorar, para la mayoría de los casos, los tiempos de las mejores alternativas actuales, tanto a la hora de contabilizar como localizar los resultados

[1]También se incluye el espacio ocupado por el uso de tablas hash para mantener los vocabularios de palabras.

[2]De acuerdo al subconjunto de XPath abordado en este trabajo.

de la consulta. Sólo en caso de que a mayores se precise su serialización, XXS no exhibe un comportamiento tan destacadao (si bien sigue siendo competitivo), ya que a diferencia de otros sistemas, que pueden acceder directamente al texto, XXS necesita recomponer los bytes que constituyen el código de una palabra y que se encuentran repartidos en los distintos nodos del XWT, antes de decodificarla y poder mostrarla al usuario. Sin embargo, en este caso, cabe mencionar una de las principales características de XXS que hacen que esas diferencias de tiempos en este escenario sean relativas: su capacidad para obtener los resultados bajo demanda, gracias al esquema de evaluación *lazy*. XXS es capaz de devolver inmediatamente (en menos de un milisegundo, en gran parte de las consultas) un primer conjunto de resultados y continuar produciendo el resto mientras los primeros son analizados por el usuario.

De otro lado, en relación al espacio, XXS es, con diferencia, la herramienta que utiliza una menor cantidad de espacio, así como la que requiere también menores tiempos de construción. Es importante notar que, en XXS, la representación comprimida (y auto-indexada) de los documentos es proporcionada a través del XWT (teniendo en cuenta el gasto de espacio adicional para mejorar su eficiencia, previamente indicado). Los experimentos demuestran que el resto de los sistemas analizados (incluyendo también compresores específicos XML con capacidades de consulta) requieren entre 2 y 5 veces más espacio que el ocupado por XXS. Esta propiedad de XXS resulta especialmente relevante a la hora de trabajar con corpus de gran tamaño que, de otra forma, deberían ser manipulados en disco.

Así pues, con este trabajo se ha puesto de manifiesto que nuestra propuesta, XXS, es una herramienta robusta y escalable, que permite responder de manera eficiente un amplio subconjunto del lenguaje de consulta XPath, utilizando para ello una cantidad de espacio mínima. Actualmente, no existe otra alternativa con características comparables, capaz de mostrar un comportamiento igual de notable en la misma cantidad de espacio.

C.4 Trabajo Futuro

Los trabajos realizados en esta tesis, así como los resultados obtenidos, hacen que algunas de las posibles líneas de investigación futuras a considerar sean las siguientes:

- Dado el buen comportamiento y propiedades exhibidas por XXS, estamos planeando ampliar el subconjunto de XPath abordado en este trabajo y tratar también algunas de sus extensiones más recientes, como desigualdades, o predicados posicionales, así como consolidar su *status* dando soporte, a mayores, al lenguaje de consultas XQuery. Si tenemos en cuenta que este último utiliza de base XPath, resulta realmente interesante poder aplicar las destacadas capacidades de consulta de nuestra herramienta a la resolución de consultas FLWOR propias del lenguaje XQuery.

- Otra importante línea de investigación futura pasa por incluir opciones de recuperación de documentos en nuestra solución, de forma que los resultados de la consulta puedan proporcionar también una información a nivel de documento. Este tipo de recuperación es adecuado para determinados escenarios, como por ejemplo, aquél donde un usuario quiere encontrar los documentos más relevantes a una consulta de entre una colección de ellos. Asimismo, también planeamos estudiar la introducción de medidas de relevancia, de manera que los resultados (bien en forma de componentes XML específicos, bien en forma de documentos) posean un grado de importancia en el contexto de las consultas, que posibilite además realizar su clasificación atendiendo a ese valor.

  Una primera aproximación a este objetivo ha sido ya estudiada, y presentada en [BCPNP12].

- Como forma de poner nuestro trabajo a disposición de los miembros de la comunidad y promover su uso, tenemos planeado proporcionar una API que permita la integración de XXS en otros sistemas como, por ejemplo, bibliotecas digitales; y también se prevé iniciar el proceso de creación de una completa aplicación para el almacenamiento comprimido y consulta de documentos XML, utilizando de base el trabajo desarrollado.

Bibliography

[ABMP07] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. XQueC: A query-conscious compressed XML database. *ACM Transactions on Internet Technology*, 7(2), 2007.

[Abr63] N. Abramson. *Information Theory and Coding*. McGraw-Hill, 1963.

[ACM$^+$10] D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyen, J. Sirén, and N. Välimäki. Fast in-memory XPath search using compressed indexes. In *Proc. of the 26th IEEE International Conference on Data Engineering (ICDE)*, pages 417–428, 2010.

[ANF07] J. Adiego, G. Navarro, and P. Fuente. Lempel-Ziv compression of highly structured documents. *Journal of the American Society for Information Science and Technology*, 58(4):461–478, 2007.

[ANF09] J. Adiego, G. Navarro, and P. Fuente. A prototype for querying over lzcs transformed documents. *IEEE Latin American Transactions*, 7(3):353–360, 2009.

[Ant97] G. Antoshenkov. Dictionary-based order-preserving string compression. *International Journal on Very Large Data Bases*, 6(1):26–39, 1997.

[BB04] D. K. Blandford and G. E. Blelloch. Compact representations of ordered sets. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–19, 2004.

[BCCN06] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based XML projection. In *Proc. of the 32th International Conference on Very Large Data Bases (VLDB)*, pages 271–282, 2006.

[BCPN09] N. R. Brisaboa, A. Cerdeira-Pena, and G. Navarro. A compressed self-indexed representation of XML documents. In *Proc.of the 13th European Conference on Digital Libraries (ECDL)*, pages 273–284, 2009.

[BCPNP12] N. R. Brisaboa, A. Cerdeira-Pena, G. Navarro, and O. Pedreira. Ranked document retrieval in (almost) no space. In *Proc. of the 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 155–160, 2012.

[BCW90] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, 1990.

[BDM+05] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BFLN08] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 139–146, 2008.

[BFLN12] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval*, 15(6):527–557, 2012.

[BFNP03] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. (s, c)-Dense Coding: An optimized compression code for natural language text databases. In *Proc. of the 10th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 122–136, 2003.

[BFNP07] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.

[BGvK+06] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 479–490, 2006.

[BHMR07] J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[BINP03] N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. R. Paramá. An efficient compression code for text databases. In *Proc. of the 25th European Conference on Information Retrieval Research (ECIR)*, pages 468–481, 2003.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[Bon02] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. PhD thesis, University of Amsterdam, Amsterdam, Netherlands, 2002.

[BR04] T. Böhme and E. Rahm. Supporting efficient streaming and insertion of XML data in RDBMS. In *Proc. of the 3rd International Workshop on Data Integration over the Web (DIWeb)*, pages 70–81, 2004.

[BSTW86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[BW94] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. In *Tech. Rep 124, Digital Equipment Corporation*, 1994.

[BYRN99] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Weslwy Longman, 1999.

[CAYBF05] E. Curtmola, S. Amer-Yahia, P. Brown, and M. F. Fernández. Galatex: a conformant implementation of the XQuery full-text language. In *Proc. of the 14th International World Wide Web Conference (WWW)*, pages 1024–1025, 2005.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007.

[Che01] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proc. of the 11th Data Compression Conference (DCC)*, pages 163–172, 2001.

[Che05] J. Cheney. An empirical evaluation of simple DTD-conscious compression techniques. In *Proceedings of the 8th International Workshop on the Web and Databases (WebDB)*, pages 43–48, 2005.

[Cla96] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Ontario, Canadá, 1996.

[CLL05] Y.-T. Chiang, C.-C. Lin, and H.-I. Lu. Orderly spanning trees with applications. *SIAM Journal on Computing*, 34(4):924–945, 2005.

[CM01] J. Clark and M Murata. RELAX NG specification. OASIS Committee, 2001.

[CN04] J. Cheng and W. Ng. XQzip: Querying compressed XML using structural indexing. In *Proc. of the 9th International Conference on Extending Database Technology (EDBT)*, pages 219–236, 2004.

[CN07] F. Claude and G. Navarro. A fast and compact web graph representation. In *Proc. of the 14th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 118–129, 2007.

[CW84] J. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communication*, 32(4):396–402, 1984.

[Deu96] P. Deutsch. Deflate compressed data format specification version 1.3, 1996.

[dMNZBY00] E. Silva de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[DOM] DOM, W3C Recommendation of Document Object Model. `http://www.w3.org/DOM`.

[DRR06] O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In *Proc. of the 5th International Workshop on Experimetal Algorithms (WEA)*, pages 134–145, 2006.

[EXI] eXist-DB Open Source Native XML Database. `http://www.exist-db.org`.

[FLMM05] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.

[FLMM06] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *Proc. of the 15th International World Wide Web Conference (WWW)*, pages 751–760, 2006.

[FLMM09] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):4:1–4:33, 2009.

[FM01] P Ferragina and G. Manzini. An experimental study of a compressed index. *Information Sciences: special issue on "Dictionary Based Compression"*, 135(1-2):13–28, 2001.

[FM05] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[FM08a] A. Farzan and J. I. Munro. Succinct representations of arbitrary graphs. In *16th Annual European Symposium (ESA)*, pages 393–404, 2008.

[FM08b] A. Farzan and J. I. Munro. A uniform approach towards succinct representation of trees. In *Proc. of the 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 173–184, 2008.

[FM11] A. Farzan and J. Ian Munro. Succinct representation of dynamic trees. *Theoretical Computer Science Journal*, 412(24):2668–2678, 2011.

[Fra06] M. Franceschet. XPathMark: Functional and performance tests for XPath. In *XQuery Implementation Paradigms*, 2006.

[FSC+03] M. F. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 1077–1080, 2003.

[Ful] W3C Recommendation of XQuery and XPath Full Text 1.0. `http://www.w3.org/TR/xpath-full-text-10`.

[Gal] Galax. `http://galax.sourceforge.net`.

[GGG+07] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *Proc. of the 15th Annual European Symposium on Algorithms (ESA)*, pages 371–382, 2007.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[GGV04] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 636–645, 2004.

[GHSV06] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. In *Proc. of the 16th Data Compression Conference (DCC)*, pages 213–222, 2006.

[GKP05] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005.

[GMOS02] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, pages 173–189, 2002.

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[GN07] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 216–227, 2007.

[Gol66] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[GP98] C. Goldfarb and P. Prescod. *The XML Handbook*. Prentice Hall, 1998.

[GRR04] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–10, 2004.

[GRRR04] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 159–172, 2004.

[GRRR06] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

[GS00] M. Girardot and N. Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the web. *Computer Networks*, 33(1-6):747–765, 2000.

[GST04] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL hosts. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 252–263, 2004.

[Gur89] E. M. Gurari. *Introduction to the theory of computation*. Computer Science Press, 1989.

[GvKT03] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 524–525, 2003.

[GvKT04] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29:91–131, 2004.

[GW97] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the 23th International Conference on Very Large Data Bases (VLDB)*, pages 436–445, 1997.

[Hea78] H. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, 1978.

[HM10] M. He and J. I. Munro. Succinct representations of dynamic strings. In *Proc. of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 334–346, 2010.

[HMR07] M. He, J. Ian Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In *Proc. of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 509–520, 2007.

[Hos10] H. Hosoya. *Foundations of XML Processing: The Tree Automata Approach.* Cambridge University Press, 2010.

[How93] P. G. Howard. The design and analysis of efficient lossless data compression systems. Technical report, Providence, RI, USA, 1993.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[JSS07] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.

[Kay08] Michael Kay. Ten reasons why Saxon XQuery is fast. *IEEE Data Engineering Bulletin*, 31(4):65–74, 2008.

[KM90] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 1(4):425–448, 1990.

[Kra49] L. G. Kraft. A device for quantizing, grouping and coding amplitude modulated pulses. Master's thesis, Department of Electrical Engineering, MIT, Cambridge, MA, USA, 1949.

[KY00] J. C. Kieffer and E. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[KYNC00] J. C. Kieffer, E. Yang, G. J. Nelson, and P. C. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46(4):1227–1245, 2000.

[Lad11] S. Ladra. *Algorithms and Compressed Data Structures for Information Retrieval*. PhD thesis, University of A Coruña, A Coruña, Spain, 2011.

[LDM05] G. Leighton, J. Diamond, and T. Müldner. AXECHOP: A grammar-based compressor for XML. In *Proc. of the 15th Data Compression Conference (DCC)*, page 467, 2005.

[LE07] C. League and K. Eng. Schema-based compression of XML data with RELAX NG. *Journal of Computers*, 2(10):9–17, 2007.

[Li03] W. Li. XComp: an XML Compression Tool. Master's thesis, University of Waterloo, Waterloo, Ontario, Canadá, 2003.

[LM00] N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722 –1732, 2000.

[LMD05] G. Leighton, T. Müldner, and J. Diamond. TREECHOP: A tree-based queriable compressor for XML. Technical report, Acadia University, 2005.

[LMR+05] J. A. List, V. Mihajlovic, G. Ramírez, A. P. de Vries, D. Hiemstra, and H. E. Blok. Tijah: Embracing IR methods in XML databases. *Information Retrieval*, 8(4):547–570, 2005.

[LNWL03] W. Y. Lam, W. Ng, P. T. Wood, and M. Levene. XCQ: XML compression and querying system. In *Proc. of the 12th International World Wide Web Conference (WWW)*, 2003.

[LS00] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 153–164, 2000.

[LW02] M. Levene and P. Wood. XML structure compression. In *Proc. of the 2nd International Workshop on Web Dynamics*, 2002.

[LY08] H.-I. Lu and C.-C. Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms*, 4(3):28:1–28:13, 2008.

[LZLY05] Y. Lin, Y. Zhang, Q. Li, and J. Yang. Supporting efficient query processing on compressed XML files. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC)*, pages 660–665, 2005.

[Mei02] W. Meier. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*, pages 169–183, 2002.

[MN07] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science Journal*, 387(3):332–347, 2007.

[MN10] S. Maneth and K. Nguyen. XPath whole query optimization. *International Journal on Very Large Data Bases*, 3(1):882–893, 2010.

[MNW98] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998.

[MNZBY98] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 298–306, 1998.

[Mof89] A. Moffat. Word-based text compression. *Software: Practice and Experience*, 19(2):185–198, 1989.

[Mof90] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communication*, 38(11):1917–1921, 1990.

[MPC03] J. Min, M. Park, and C. Chung. XPRESS: A queriable compression for XML data. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 122–133, 2003.

[MR97] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[MR01] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[MR04] J. I. Munro and S. S. Rao. Succinct representations of functions. In *Proc. of the 31th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1006–1015, 2004.

[MRR01] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[MS03] A. Marian and J. Siméon. Projecting XML documents. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 213–224, 2003.

[MT02] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002.

[Mun96] J. I. Munro. Tables. In *Proc. of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

[MWA+98] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical Report 1998-46, Stanford InfoLab, 1998.

[NBY95] G. Navarro and R. Baeza-Yates. A language for queries on structure and contents of textual databases. In *Proc. of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 93–101, 1995.

[NLWL06] W. Ng, W. Y. Lam, P. T. Wood, and M. Levene. XCQ: A queriable XML compression system. *Knowledge and Information Systems*, 10(4):421–452, 2006.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[NMW97a] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3):103–116, 1997.

[NMW97b] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7(1):67–82, 1997.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings - Practical on-line search algorithms for text and biological sequences*. Cambridge University Press, 2002.

[Olt07] D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Transactions on Knowledge and Data Engineering*, 19(7):934–949, July 2007.

[OMFB02] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops*, pages 109–127, 2002.

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc.of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[Pag99] R. Pagh. Low redundancy in static dictionaries with O(1) worst case lookup time. In *Proc. of the 26th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 595–604, 1999.

[PC05] F. Peng and S. S. Chawathe. XSQ: A streaming XPath engine. *ACM Transactions on Database Systems*, 30(2):577–623, 2005.

[Qiz] XML Mind products. Qizx XML database engine. `http://www.xmlmind.com/qizx`.

[RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[RSF06] C. Re, J. Siméon, and M. F. Fernández. A complete and efficient algebraic compiler for XQuery. In *Proc. of the 22nd International Conference on Data Engineering (ICDE)*, page 14, 2006.

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[Sad07] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[Sak09] S. Sakr. XML compression techniques: A survey and comparison. *Journal of Computer and System Sciences*, 75(5):303–322, 2009.

[SAXa] SAX. `http://www.saxproject.org`.

[Saxb] Saxon: The XSLT and XQuery Processor. `http://saxon.sourceforge.net`.

[Sch] Schematron. A language for making assertions about patterns found in XML documents. `http://www.schematron.com`.

[SGS08] P. Skibinski, S. Grabowski, and J. Swacha. Effective asymmetric XML compression. *Software: Practice and Experience*, 38(10):1027–1047, 2008.

[Shk02] D. Shkarin. PPM: One step to practicality. In *Proc. of the 12th Data Compression Conference (DCC)*, pages 202–211, 2002.

[SK64] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

[SM02] N. Sundaresan and R. Moussa. Algorithms and programming models for efficient representation of XML for internet applications. *Computer Networks*, 39(5):681–697, 2002.

[SN10] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. of the 21th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.

[SPE] SPEX: XPath Evaluation against XML streams. `http://spex.sourceforge.net`.

[SS05] H. Subramanian and P. Shankar. Compressing XML documents using recursive finite state automata. In *Proc. of the 10th International Conference on Implementation and Application of Automata (CIAA)*, pages 282–293, 2005.

[SSK07] M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *Proc. of the 23rd International Conference on Data Engineering (ICDE)*, pages 236–245, 2007.

[SW49] C. E. Shannon and W. Weaver. *A Mathematical Theory of Communication.* University of Illinois Press, 1949.

[TH02] P. M. Tolani and J. R. Haritsa. XGring: A query-friendly XML compressor. In *Proc. of the 18th International Conference on Data Engineering (ICDE)*, pages 225–234, 2002.

[TM97] A. Turpin and A. Moffat. Fast file search using text compression. In *Proc. of the 20th Australian Computer Science Conference (ACSC)*, pages 1–8, 1997.

[Tom03] V. Toman. Compression of XML data. Master's thesis, Charles University, Prague, Czech Republic, 2003.

[TVB+02] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 204–215, 2002.

[Wan03] R. Wan. *Browsing and Searching Compressed Documents*. PhD thesis, Dept. of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, 2003.

[WB91] I. H. Witten and T. C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, 1991.

[Wel84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[WLLH04] H. Wang, J. Li, J. Luo, and Z. He. XCPaqs: Compression of XML documents with XPath query support. In *International Conference on Information Technology: Coding and Computing (ITCC)*, pages 354–358, 2004.

[WLS07] R. K. Wong, F. Lam, and W. M. Shui. Querying and maintaining a compact xml storage. In *Proc. of the 16th International World Wide Web Conference (WWW)*, pages 1073–1082, 2007.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[Xal] The Apache Xalan Project. `http://xalan.apache.org`.

[XDM] XDM, W3C Recommendation of XQuery 1.0 and XPath 2.0 data model. `http://www.w3.org/TR/xpath-datamodel`.

[XLi] XLink, W3C XML Linking Language 1.0. `http://www.w3.org/TR/xlink`.

[XMLa] XML 1.0, W3C Recommendation of Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/REC-xml`.

[XMLb] XMLZip - XML Solutions. `http://www.xmls.com`.

[XPaa] XPath 1.0, W3C Recommendation of XML Path Language (XPath) 1.0. `http://www.w3.org/TR/xpath`.

[XPab] XPath 2.0, W3C Recommendation of XML Path Language (XPath) 2.0. `http://www.w3.org/TR/xpath20`.

[XPo] XPointer, W3C XML Pointer Language. `http://www.w3.org/TR/xptr`.

[XQu] XQuery 1.0, W3C Recommendation of XML Qquery Language 1.0. `http://www.w3.org/TR/xquery`.

[XSD] XSD, W3C XML Schema Definition Language 1.1. `http://www.w3.org/XML/Schema`.

[XSL] XSLT, W3C Recommendation of XSL Transformations. `http://www.w3.org/TR/xslt`.

[XSQ] XSQ: A Streaming XPath Engine. `http://www.cs.umd.edu/projects/xsq`.

[XUp] XUpdate, W3C Recommendation of XQuery Update Facility 1.0. `http://www.w3.org/TR/xquery-update-10`.

[Zip49] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley (Reading MA), 1949.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.