IMPLEMENTATION OF AN EFFICIENT XML FILTERING MECHANISM WITH XPATH EXPRESSIONS BASED ON XTRIE

by

Sarafis Dimitrios

A thesis submitted in fulfillment of the
Requirements for the degree of

ELECTRONIC AND COMPUTER ENGINEERING

TECHNICAL UNIVERSITY OF CRETE
DEPARTMENT OF ELECTRONIC AND COMPUTER ENGINEERING

Chania 2004

Abstract

Recently the *publish/subscribe* model has been popular in many application domains due to its efficiency on the integration process. Early publish/subscribe systems have relied on typical matching schemes, such as simple comparison predicates on attribute values The emergence of XML as a standard for information exchange on the Internet has led to an increased interest in using more expressive subscription mechanisms based on XPath expressions. In the last few years several systems have addressed the problem of filtering a stream of XML documents.

This dissertation presents the implementation of a novel index structure, termed XTrie, which supports the efficient filtering of XML documents with XPath expressions. In this work, we dealt with the problem of filtering streaming XML documents with XPath expressions. The main goal was to implement the XTrie structures and algorithms for efficient XML filtering. The XTrie system, provides efficiency and scalability, has low space requirements and offers high throughput. Those features make it especially attractive for large scale distributed systems over the internet such as publish/subscribe systems.

Our implementation of XTrie supports the ordered matching model and XPath expressions with wildcards, predicates, and closures. We believe that we capture the key features of XPath expressions that will prove most useful in data-dissemination applications.

We use the Java 2 SDK 1.4 and the herces SAX parser interface for our implementation.

Contents

1 Introduction 7
1.1 Overview 7
1.2 Contributions of the Dissertation 9
1.3 Organization of the Dissertation 11

2 Background 12
2.1 Data Models and Query Languages 12
2.1.1 Structured Data Model 13
2.1.2 Unstructured Data Model 13
2.1.3 Semi-structured Data Model 13
2.2 eXtensible Markup Language (XML) 14
2.2.1 XML Data 14
2.2.2 DTD 16
2.2.3 Well-Formedness and Validity 17
2.2.4 XML Tree 18
2.3 XPath Language 18
2.3.1 XPath Expressions (XPE's) 19
2.4 XML Parsers 21
2.4.1 Document Object Model (DOM) Parser 21
2.4.2 Simple API for XML (SAX) Parser 21
2.5 Summary 23

3 Related work 24
3.1 Streaming vs. non-streaming XML Data 24
3.2 Systems for Filtering Streaming XML Data 25
3.3 Systems for Querying XML Data 27
3.3.1 Streaming XML Data 27
3.3.2 Non-streaming XML Data 29
3.4 Other Systems 31
3.5 Summary 33

4 XPE Decompositions and Matchings 34
4.1 XPE-Tree and XPE Matching 34
4.2 Substring Decompositions 36
4.2.1 Minimal Decomposition 37

4.2.2 Simple Decomposition 38
4.3 Substring-Trees 39
4.4 Matching with Substrings 40
4.4.1 Partial and Complete Matching 41
4.4.2 Redundant Matching 41
4.5 Subtree Matching 42
4.6 Summary 44

5 XTrie Structure and Indexing Scheme 46
5.1 The Index Structure 46
5.1.1 The Substring-Table 47
5.1.2 The Trie 48
5.2 The XTrie Matching Algorithm 49
5.2.1 B-array 50
5.2.2 C-array 50
5.2.3 Detect non-redundant Matchings 51
5.2.4 Details of Matching Algorithm 53
5.2.5 Theoretical Space and Time Complexity 57
5.3 Attributes and Text Data 57
5.4 Maintenance 58
5.4.1 Reverse Trie 58
5.4.2 Insertions 59
5.4.3 Deletions 61
5.5 Lazy XTrie Optimization 63
5.6 Summary 67

6 System Architecture and Implementation 69
6.1 XTrie Architecture 69
6.2 XTrie Implementation 70
6.2.1 XPath Parser 70
6.2.2 XPE Tree Constructor 72
6.2.3 Substring Tree Constructor 73
6.2.4 XTrie Builder 74
6.2.5 XML Document Parser 76
6.2.6 XTrie Matching Algorithms 77
6.3 Summary 79

7 Conclusions and Future work 80
7.1 Concluding Remarks and Summarizing 80
7.2 Future Work 82

List of Figures

Figure 1.1 : The Publish-Subscribe system architecture..8
Figure 1.2 : XTrie architecture..9
Figure 2.1 : A simple XML Document example..15
Figure 2.2 : The DTD of our XML document..17
Figure 2.3 : The XML Tree of our XML document..18
Figure 2.4 : Our XPath fragment..19
Figure 2.5 : A more complex XML document..20
Figure 2.6 : The XML DOM Tree of our XML Document..22
Figure 4.1 : XPath Tree matching with an XML Tree..35
Figure 4.2 : An XPE Tree and an XML Tree example..36
Figure 4.3 : Minimal and Simple XPE decompositions..38
Figure 4.4 : Substring tree for S_b..39
Figure 4.5 : Partial and redundant matching example..42
Figure 4.6 : Example of redundant matching..43
Figure 4.7 : Subtree-based conditions for redundant matchings..44
Figure 5.1 : An XTrie example..48
Figure 5.2 : Propagation of subtree-matchings example..51
Figure 5.3 : Algorithm to search XTrie..54
Figure 5.4 : Algorithm to process a matched substring..55
Figure 5.5 : Algorithm to update run-time information arrays and detect complete matchings..56
Figure 5.6 : Algorithm for insertion of a set of XPEs..60
Figure 5.7 : Algorithm for XPE deletion..62
Figure 5.8 : Algorithm to search lazy XTrie..64
Figure 5.9 : Algorithm to process a matching substring in lazy XTrie..65
Figure 5.10 : Auxiliary algorithm to process a matching substring in lazy XTrie......66
Figure 6.1 : The XTrie Architecture..70
Figure 6.2 : Location steps produced by the XPath parser..71
Figure 6.3 : (a) XPE Tree node structure and (b) XPE Tree..72
Figure 6.4 : (a) Substring Tree node structure and (b) Substring Tree..73
Figure 6.5 : (a) Trie node structure and (b) ST Row structure..74
Figure 6.6 : The input XPE set..75
Figure 6.7 : The Trie and the Substring Table produced by XTrie builder..76
Figure 6.8 : The XML Document as input of the XTrie system..78
Figure 6.9 : The output of the XTrie system..79

List of Tables

Table 2.1 : XML SAX events 23
Table 5.1 : XTrie basic notations...47
Table 5.2 : Execution trace of changes to *B* array for the matching of *p* on *D* in Figure 4.1...52

Chapter 1

1 Introduction

Recently the *publish/subscribe* model has been popular in many application domains due to its efficiency on the integration process. This model allows *publishers* to selectively disseminate data to a large number of widely dispersed *subscribers* who have registered their interest in specific information items. Early publish/subscribe systems [1, 8, 37] have relied on a typical scheme that implements matching mechanisms based on combinations of keywords and predicates over associative values of the keywords. The emergence of XML [42] as a standard for information exchange on the Internet has led to an increased interest in using more expressive subscription mechanisms (e.g., based on XPath [46] expressions) that exploit both the structure and the content of published XML documents. There are several data-filtering mechanisms, which are used to effectively identify the subscription profiles that match an incoming XML document. This dissertation presents the implementation of a novel index structure, termed *XTrie* [9], which supports the efficient filtering of XML documents based on XPath expressions.

1.1 Overview

Recently, advanced services for providing rapid notifications of certain events, have been deployed in the form of information dissemination in many domains including stock quotes, financial news, transportation and so on. This trend has led to the

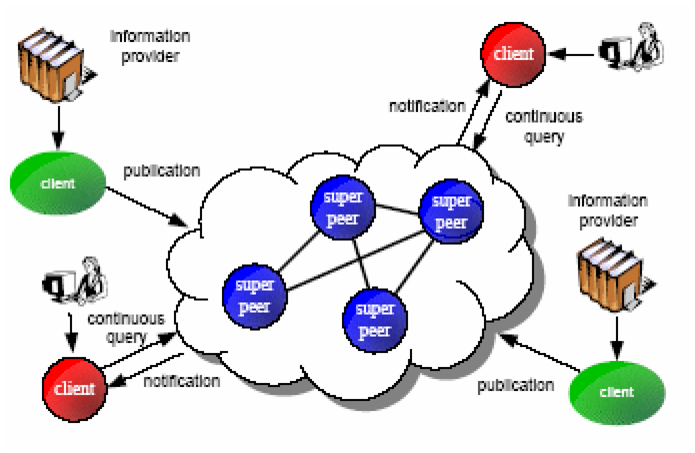


Figure 1.1 : The Publish-Subscribe system architecture

emergence of novel middleware architectures that asynchronously propagate data from a set of *publishers* (i.e., data generators) to a large number of widely dispersed *subscribers* (i.e., data consumers), who have pre-registered their interest in specific information items [8]. In general, such *publish-subscribe* architectures are implemented using a set of networked servers that selectively propagate relevant messages to the consumer population, where message relevance is determined by *subscriptions* representing the consumers' interests in specific messages. An example of the publish/subscribe architecture is shown in Figure 1.1. The majority of existing publish/subscribe systems [1, 8, 37] have typically relied on simple subscription mechanisms, such as keyword or "bag of words" matching, or simple comparison predicates on attribute values. For example, systems such as Gryphon [1], Siena [8], and Elvin [37], all use filters in the form of a set of attributes and simple arithmetic or Boolean comparisons on the values of these attributes. Recent trends of XML [42] have rapidly been increasing areas where XML data need to be analyzed for further processing such as information exchanges on the Internet. Therefore, environments where XML data from multiple information providers are streamed and various users define their interest over such XML data are frequently referred to imply necessity of XML based publish/subscribe system architectures. In such environments both contents and structures are used to match XML data against users' interests, so a language for describing the interests requires expressing desirable contents in specific structures. For this purpose, XPath [46] which is a W3C [43] proposed standard for

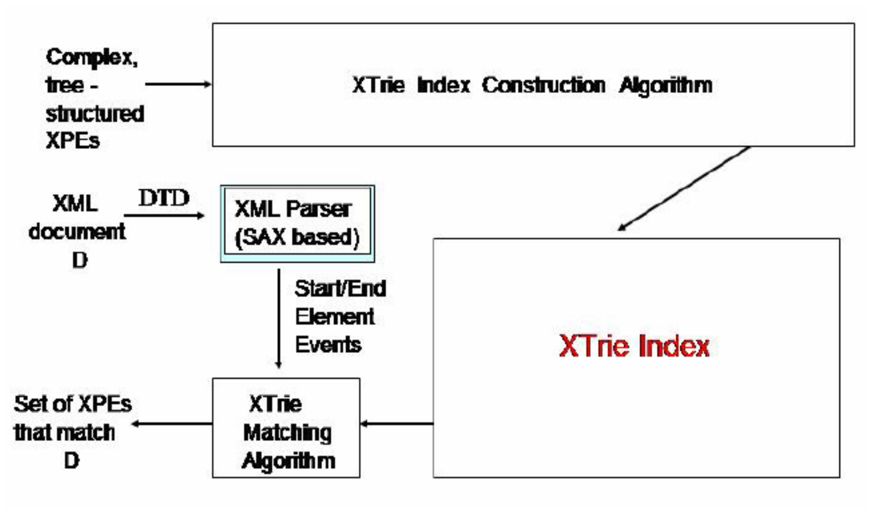


Figure 1.2 : XTrie architecture

addressing parts of an XML document, has been adopted as a filter-specification language by a number of recent XML data dissemination systems (e.g., XFilter [2], Intel's NetStructure XML Accelerator [21]). Due to the importance of the effective identification of subscriptions that match an incoming XML document we have implemented an efficient XML matching scheme, named XTrie [9]. More specifically, XTrie is able to solve the key problem [9] faced in XPath based data-dissemination systems, which can be abstracted as follow:

"Given a large collection *P* of XPath expressions (XPEs) and an input XML document *D*, find the subset of XPEs in *P* that match *D*."

The key technique which is used by XTrie for expediting XPE retrieval is to construct an appropriate index structure on the given collection of XPE subscriptions, as shown in Figure 1.2. Furthermore, simplistic approaches (e.g., building an index based solely on the element names contained in the XPEs) can result in very ineffective retrieval schemes that incur a lot of unnecessary checking of (irrelevant) XPE subscriptions.

1.2 Contributions of the Dissertation

In this dissertation, we implement a novel index structure, termed *XTrie*, which supports the efficient filtering of XML documents based on XPath expressions. We implement several important features of XTrie indexing scheme which are especially attractive for large-scale publish/subscribe systems. Thus, our implementation of XTrie:

- Supports effective filtering based on complex XPath expressions (as opposed to simple, single-path specifications).
- Supports XPath expressions with wildcards and closures.
- Allows for predicates comparing element/attribute values against constants.
- Supports ordered matching of XML data. Note that ordered matching is an important requirement for many applications (e.g., document processing) that has typically been overlooked in existing data dissemination systems.
- Provides extremely efficient filtering by indexing on a carefully-selected set of substrings [9] (rather than individual element names) in the XPEs and using a sophisticated matching algorithm [9]. So XTrie is able to minimize both the number and the cost of the required index probes.
- Supports on-line filtering of streaming XML data, based on the event-based SAX parsing interface [33] (in contrast to the alternative DOM parsing interface [41], which requires a main-memory representation of the XML data tree to be built before filtering can commence).

Informative, the only other SAX-based index structure for the XPE retrieval problem is Altinel and Franklin's XFilter [2], which relies on indexing the XPE element names using a hash-table structure. By indexing on substrings rather than individual element names, the XTrie index provides a much more effective indexing mechanism than XFilter. A further limitation of XFilter is that its space requirement can grow to a very large size as an input document is parsed, which can also increase the filtering time significantly. The experimental results [9] over a wide range of XML document and XPath expression workloads, demonstrate that the XTrie index scheme scales well to high volumes of XPEs and complex documents, and consistently outperforms XFilter by significant margins (factors of up to one or two orders of magnitude).

1.3 Organization of the Dissertation

This dissertation is organized as follows. In Chapter 2, we give an overview of the general data models and languages and we present the basic features of the XML data model and XPath query language. In Chapter 3, we briefly discuss alternative mechanisms for XML data processing. Chapter 4 presents some important definitions about the basic features of the XTrie indexing scheme. Then, in Chapter 4, we present the methodology for decomposing complex XPEs into *substrings* for effective indexing. In Chapter 5, we present the XTrie index structure and algorithms. Also, Chapter 5 discusses an optimized variant of XTrie. This variant is optimized to further reduce the number of unnecessary index probes. Chapter 6 presents the basic data structures in our XTrie implementation. Finally, in Chapter 7, we present our conclusions and future work possibilities.

Chapter 2

2 Background

In this chapter we present the basic data models and their query language. We discuss in detail the XML features and the XPath language. Also we present some examples to show the fragment of XPath which is used in the dissertation. Final, we discuss several important features of the XML data model such as: XML syntax [42], DTD [42], SAX [33] parser, DOM [41] parser, and XML validation [42] and well-formedness [42].

2.1 Data Models and Query Languages

In the last few years, various data models have been developed. Those models, are mainly used for the representation of information and the exchange of data over the internet. Each data model consists of data, a query language and an optional schema. The first contains data values, the second is used to pose queries over those data and the latter is used to describe those data. There are three different types of data models, which are used widely in publish/subscribe systems: structured data model, unstructured data model and Semi-structured data model. In the next sections we briefly discuss those models.

2.1.1 Structured Data Model

Data in relational databases is *structured*. It has a schema (e.g. E-R diagram [39]), which is usually stored in a part of the database called the catalog, while the data values are stored separately, in tables, following a layout that is completely described by the schema. Queries, expressed in SQL [39], refer both to the schema components, such as relation names and their attributes, and to the data values, in the form of equality predicates, inequality predicates, or string matches. Research on query processing has focused on join processing techniques, join ordering, and indexes.

2.1.2 Unstructured Data Model

Text documents are *unstructured*. There is no schema, only the text, and the data consists of some large collection of documents. A query consists of a regular expression, often as simple as a single word, and the answer consists of the set of text documents that match the given query. Indexes are used here too, and they are conceptually similar to, although technically different from those in relational databases (e.g. inverted files vs. B+-trees). Research on query processing has focused, among other things, on how to process efficiently regular expressions on a text document, and has produced celebrated results such as Knuth-Morris-Pratt's [27] string search algorithm, suffix trees [27], and suffix arrays [27]. These techniques have often been based on, and even expanded automata theory.

2.1.3 Semi-structured Data Model

A new kind of data is *semi-structured data*. Although considered in one form or another for a long time, semi-structured has gained main-stream acceptance only recently, since the introduction of XML [42]. Like in structured data, we have schema components (the tags and attributes in XML), and data values are organized along these components. In this case, the schema is embedded with the data. Thus each data item can describe its own local schema. This allows much more freedom in designing the structure, and often leads to structures that were explicitly disallowed in the relational data, such as nested collections, multiple or missing subelements, elements of the same type but with different structures, heterogeneous collections, etc. In the past, researchers have studied instances of data that we would call today semi-structured, either in the form of SGML [45] documents, or as *structured documents*, i.e. documents with a predefined grammar.

2.2 eXtensible Markup Language (XML)

XML (eXtensible Markup Language) [42] is a format for specifying structured documents and data. XML is called extensible because it allows users to define their own schema, unlike HTML [44] which is a pre-defined markup language. With XML one can define his own customized markup language to describe different types of documents. XML has become a popular way to display and distribute structured data on the web because of its flexibility.

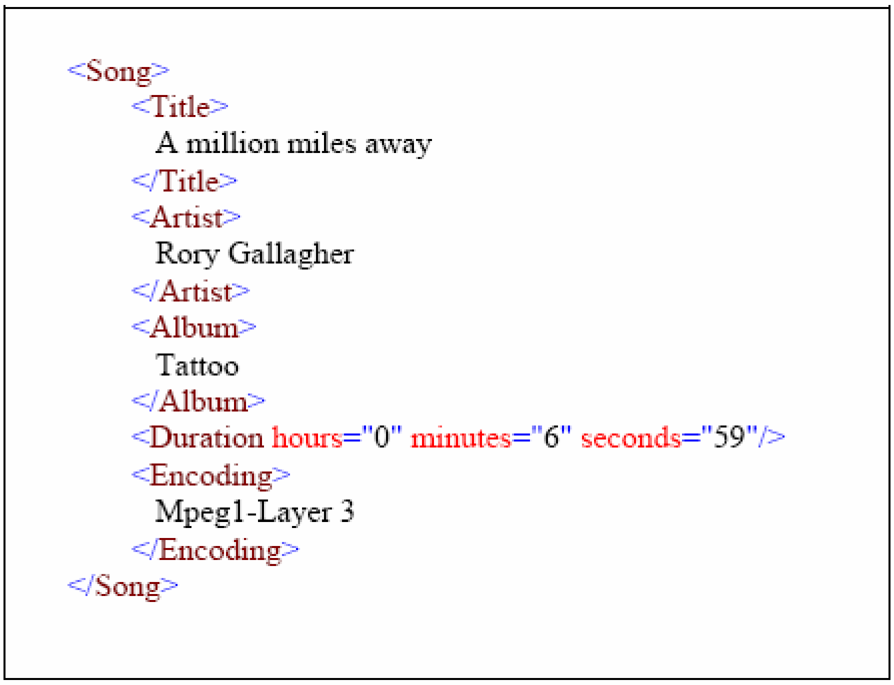
An XML specification defines a standard way to add mark-up language to documents, identifying the embedded structures in a consistent way. By applying a consistent identification structure, data can be shared between different systems, up and down the levels of agencies, across the nation, and around the world, with the ease of using the Internet. In other words, XML lays the technological foundation that supports interoperability.

XML is compatible with major Internet transmission protocols, and is also highly compressible for faster transmission. Almost all major software vendors fully support the general XML standard. XML is very developer-friendly, yet ordinary users with no particular XML expertise can make sense of an XML file. The XML standard is designed to be independent of vendor, operating system, source application, destination application, storage medium (database), and/or transport protocol.

XML is a hierarchical data model consisting of two parts: the *schema*, and the *data*. In XML, the *schema* describes the structure of the *data*. The following two sections present those parts.

2.2.1 XML Data

XML documents contain structured plain text. Authors indicate the structure by placing special text, called markup *tags,* around the text data. The structural delimiters



```
<Song>
     <Title>
       A million miles away
     </Title>
     <Artist>
       Rory Gallagher
     </Artist>
     <Album>
       Tattoo
     </Album>
     <Duration hours="0" minutes="6" seconds="59"/>
     <Encoding>
       Mpeg1-Layer 3
     </Encoding>
</Song>
```

Figure 2.1 : A simple XML Document example

are tags, which begin and end with angle brackets <...>. The text between the angle brackets contains information about the *element*; at a minimum, it names the element. An element consists of an opening tag, the element's contents, and a closing tag. Closing tags have the same name as the opening tag, but start with </. Elements can contain text, other elements, or a mixture of the two; elements can also be empty tags, are strictly case-sensitive, elements must always be closed, and cross-nesting is illegal. Empty elements have either a closing </element> tag with no contents, or are written <element/> in order to distinguish them from illegal unclosed elements. XML comments appear between the characters <!-- and --> and are usually ignored by processing applications. Tags may also contain additional information called *attributes*. The attributes are placed in the element's opening tag, and are written in the form *name="value"*.

A simple example of an XML document with information about an mp3 song file shown in Figure 2.1. The document element is <Song> which has five children elements: <Title>, <Artist>, <Album>, <Duration> and <Encoding>. The <Title> element encloses text data ("A million miles away") with information about the title of the song. The <Artist> element encloses text data ("Rory Gallagher") with information about the artist name. The <Album> element encloses text data ("Tattoo") with information about the album name. The <Duration> element has three attributes (hours, minutes, seconds) with some values representing the duration of the song. Finally the 5th child element of the document element (<Encoding>) encloses text data ("Mpeg1-Layer 3") with information about the encoding type of the mp3 file.

2.2.2 DTD

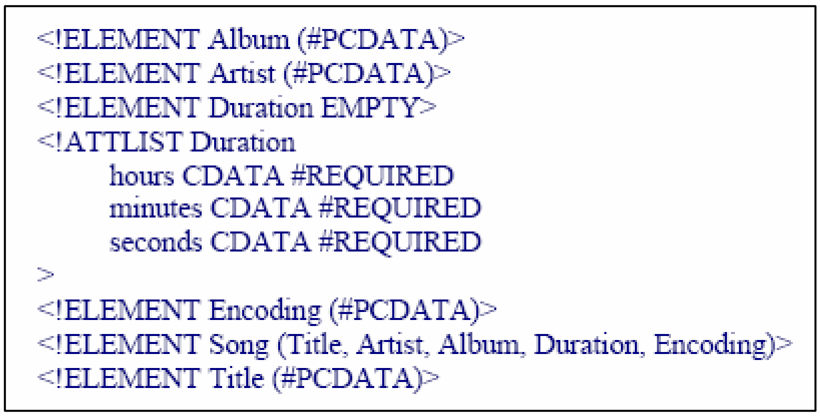
A popular language used for XML schemas definitions is DTD (Document Type Definition) [42]. DTD is a grammar that specifies a set of element types, a model for their content, constraints for the value of the attributes, and possibly declares a set of entities that can be referenced. In DTD, elements and attributes defined by the keywords <!ELEMENT> and <!ATTRIBUTE> respectively. Elements are the main building blocks of XML Schema Data documents. Once Elements are defined for a given XML document, elements can be marked up by tags. Attributes provide extra information about elements and are placed inside element tags and come in name/value pairs. The following is the general BNF syntax for element and attribute components:

```
<!ELEMENT> <elem-name> <elem-content-model>
<!ATTLIST> <attr-name> <attr-type> <attr-option>
```

The DTD specifies the attributes allowed for each element, with an indication about their content (character data, CDATA) and if the attribute must be always specified (#REQUIRED), or if it is optional (#IMPLIED). An XML document declares the DTD by a DOCTYPE declaration after the prologue but before the root element:

```
<!DOCTYPE function "functions.dtd" [...]>
```

The part within square brackets is optional, and can be used to declare further entities. The name function in the DOCTYPE declaration refers to the root element of the document, whereas functions.dtd refers to the resource containing the DTD.



```
<!ELEMENT Album (#PCDATA)>
<!ELEMENT Artist (#PCDATA)>
<!ELEMENT Duration EMPTY>
<!ATTLIST Duration
        hours CDATA #REQUIRED
        minutes CDATA #REQUIRED
        seconds CDATA #REQUIRED
>
<!ELEMENT Encoding (#PCDATA)>
<!ELEMENT Song (Title, Artist, Album, Duration, Encoding)>
<!ELEMENT Title (#PCDATA)>
```

Figure 2.2 : The DTD of our XML document

The element content model is used to define the general structure of the element. Regular expressions can be used to describe the cardinality of parts of an element such as:

- "?" (0 or 1 instance)
- "*" (0 or many instances)
- "+" (1 or many instances).

For example, the following XML element describes a paper with one title sub element, one or more author sub-element, and zero or more citation sub-elements:

```
<!ELEMENT paper (title, author+, citation*)>
```

Once the schema has been defined, the remainder of the XML document contains data. Each data element has a starting and ending tag defined by the schema. Figure 2.2 shows the DTD of the XML document in Figure 2.1.

2.2.3 Well-Formedness and Validity

XML files must match both the XML syntax and the rules given in the DTD. There are two different notions of correctness that correspond to these constraints. One is called *well-formedness* [42], and means that the XML file has a valid syntax. Moreover, XML does not define any set of predefined element types.

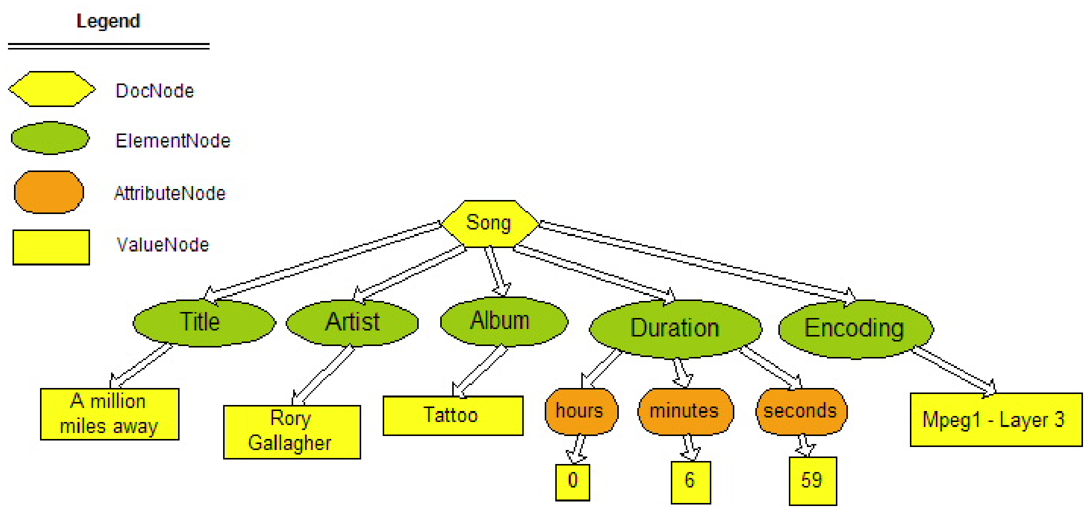


Figure 2.3 : The XML Tree of our XML document

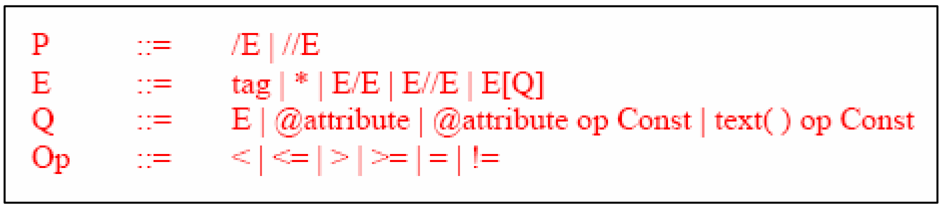
As a consequence, the document well-formedness property does not enforce any constraint on the used element types, as long as the element tags obey to the proper nesting condition. The other constraint is *validity* [42], which can only be determined by checking if the elements and attributes conform to the DTD. Validity is a stronger constraint than well-formedness, because a valid file must also be well-formed.

2.2.4 XML Tree

Every XML document can be represented as an ordered rooted tree. The root node of the *XML-tree* is the document element. Every node consists of the element name, the attributes with their values and the enclosed text between the start and the end tag. The XML-tree has a node level which represents the depth of the tree. The root node has level 1.The level of a node *d* is the level of it's father node *d'* plus 1 *(level*(*d*) = *level*(*d'*) +1). Figure 2.3 depicts the XML tree of the XML document in Figure 2.1.

2.3 XPath Language

The primary purpose of XPath is to address parts of an XML document. In support of this primary purpose, it also provides basic facilities for the manipulation of strings, numbers and Booleans. XPath uses a compact, non-XML syntax to facilitate the use of XPath within web pages and XML attribute values. XPath operates on the abstract,



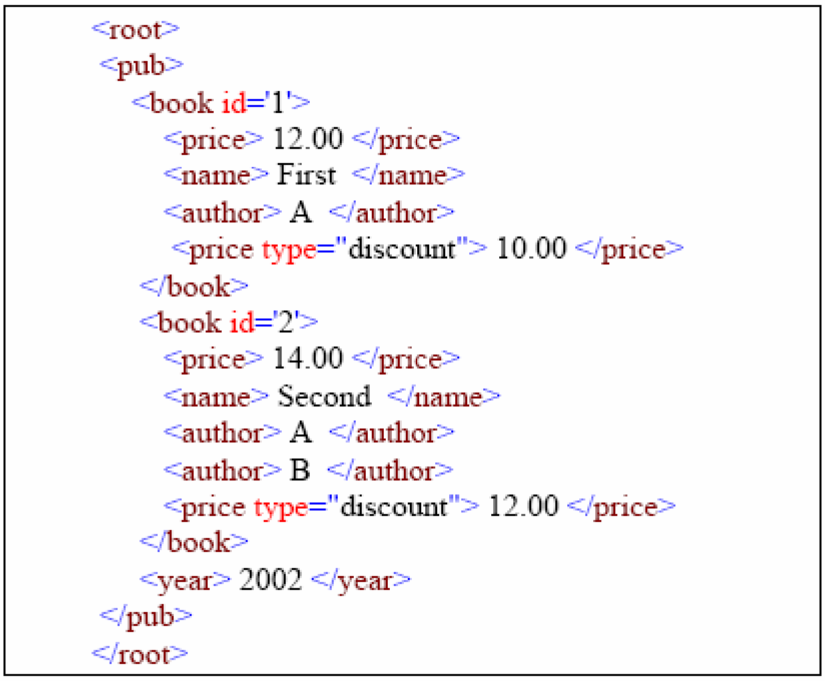
```
P     ::=   /E | //E
E     ::=   tag | * | E/E | E//E | E[Q]
Q     ::=   E | @attribute | @attribute op Const | text( ) op Const
Op    ::=   < | <= | > | >= | = | !=
```

Figure 2.4 : Our XPath fragment

logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in web pages for navigating through the hierarchical structure of an XML document.

2.3.1 XPath Expressions (XPE's)

Generally an XPath query is an expression (XPE) of the form of $N_1\ N_2\ \ldots\ N_n\ /\ O$, which consists of a location path, $N_1 N_2\ \ldots\ N_n$, and an output expression O. Each location step N_i in the location path is in the form */a :: n[p]* where *a* is an axis, *n* is a node test, and *p* is an optional predicate which is specified syntactically using square brackets. A location step matches a node in the document tree. The evaluation of an XPE yields an object whose type can be a node-set, a Boolean, a number, or a string. For the XPE retrieval problem described in [9], an XML document matches an XPE when the evaluation result is a non-empty node set. The XPath expressions don't have an output expression because we need to filter, not query XML documents. Figure 2.4 shows the simplified grammar used in our descriptions in this dissertation. The *axis* specifies the relation between the previous node and the current node. In the simplified grammar, / is shorthand for the */child :: axis*, which selects the children of the current node. Similarly, *//,* called closure axis, is shorthand for the */descendant-or-self :: node()/ axis*, which selects the current node and its descendants. If no axis is specified, the default axis is the *child axis*. However, if the axis before the first location step is omitted, the default axis is the closure axis, not the *child axis*. An element matches a location path if the path from the document root to that element, matches the sequence of labels in the location path, and satisfies all predicates. The predicates can be applied to the text or the attributes of the addressed elements, and may also include other path expressions. Any relative paths in a predicate expression are evaluated in the context of the element nodes addressed in the location step at



```
<root>
 <pub>
    <book id='1'>
       <price> 12.00 </price>
       <name> First  </name>
       <author> A  </author>
        <price type="discount"> 10.00 </price>
     </book>
     <book id='2'>
       <price> 14.00 </price>
       <name> Second  </name>
       <author> A  </author>
       <author> B  </author>
       <price type="discount"> 12.00 </price>
     </book>
     <year> 2002 </year>
 </pub>
</root>
```

Figure 2.5 : A more complex XML document

which they appear. XPath also allows the use of a wildcard operator ("*") to match any element name at a location step. The following queries, evaluated on the data of Figure 2.1, illustrate some of the key features of XPath which are used in our dissertation.

- *//Artist[text()="Rory Gallagher"]*: This query returns the Artist elements that have a text with value "Rory Gallagher". The first location step is //Artist, which consists of the closure axis //, and the node-test Artist; it has a predicate which requires the enclosed text in Artist element to has a "Rory Gallagher" value. This location step matches all descendants of the document root that have tag Artist. We note that this query may also be expressed as *Artist[text()="Rory Gallagher"],* because a missing axis in the first location step defaults to closure.

- */Song//Duration[@minutes<'7'][@seconds='59']*: This query returns the Duration elements with two attributes (minutes with value less than 7, and seconds with value 59), that have Song ancestor that occurs at the top level. We note that the second location step in this query use the closure axis.

- */Song[Title][Artist]/Encoding[text()="Mpeg1-Layer 3"]*: This query returns the Encoding elements that have a text value "Mpeg1-Layer 3". The Encoding element must have a Song parent that occurs at the top level. Also the Song element must have two other children (Title and Artist).We note that this query does not use the closure axis.

2.4 XML Parsers

To retrieve the data of a XML document, the program has to parse it. Fortunately, Java [23] already provides XML parser classes, which are described later on. Generally, there are two main technologies: DOM [41] and SAX [33]. In the next two sections we discuss those parsers.

2.4.1 Document Object Model (DOM) Parser

In Document Object Model (DOM) [41], an XML document is modelled as a node-labelled tree. Each element in the document is mapped to a subtree in the tree, whose root node is labelled with the tag of the element. Although element *E* is mapped to a subtree of the DOM tree, it is convenient to refer to the root of this subtree as the node *E*. The sub elements of an element *E* are mapped to sub elements of the node *E* that have node type of element. The attributes and text contents of element *E* are also mapped to sub elements of node *E*, but with node types *Attr* and *Text*, respectively. Figure 2.6 depicts the DOM tree of the XML document in Figure 2.5. In the figure, the nodes with dotted boxes are *Attr* nodes and the nodes without boxes are *Text* nodes.

2.4.2 Simple API for XML (SAX) Parser

For streaming data, building a DOM tree in memory is not usually desirable because the data may be unbounded. Further, we may not need the entire DOM tree to process the given query. Therefore, streaming data is better modelled using the SAX (Simple

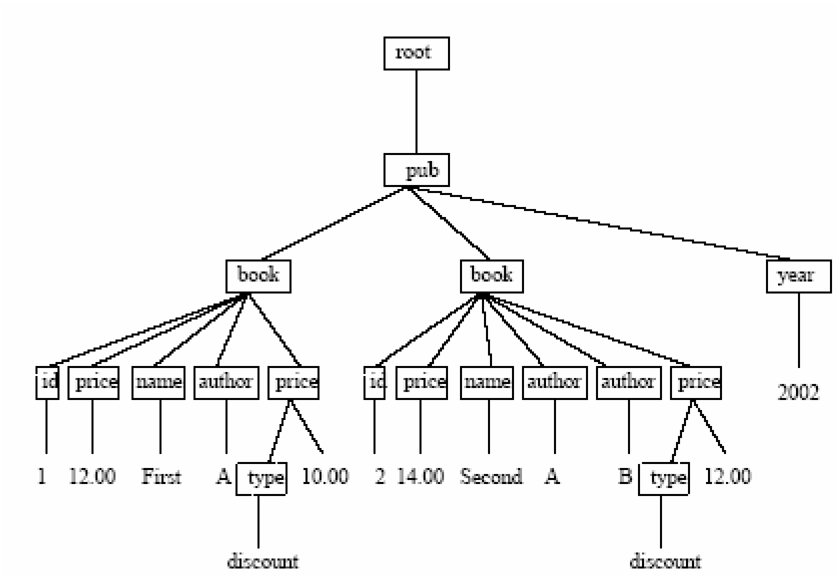


Figure 2.6 : The XML DOM Tree of our XML Document

API of XML) model [33]. Parsers based on the SAX Application Programming Interface process an XML document and generate a sequence of SAX events. For each opening and closing tag of an element, the SAX parser generates, respectively, a begin and end event. The begin event of an element comes with an attribute list that encodes the names and values of attributes associated with the element. (Since the XML standard does not allow an element to be associated with multiple attributes with the same name, this list is composed of pairs that are uniquely identified by their first element). The text contents enclosed by the opening and closing tag result in the SAX parser generating a text event. Essentially, the sequence of the SAX events corresponds to a pre-order traversal of the DOM tree of the data in which the attribute nodes are combined with their parents. Table 2.1 depicts the SAX events generated by a SAX parser given the data of Figure 2.5 as input.

In more detail, we model the input as a sequence of SAX events, where each event is a three tuple (tag, attrs, type,) where:

- *tag* is a string that corresponds the name of the element that generates the SAX event.
- *attrs* is the attribute list of this element. That is, it is a list of elements of the form (*a*, *v*) indicating that the element has attribute *a* with value *v*. Recall that, since elements do not have multiple attributes with the same name, there is

Event	Description
(root, φ, B)	the begin event of root element.
(pub, φ, B)	the begin event of pub element.
(name, {(id, "1")}, B)	the begin event of book element. The name-value list {(id,"1")} is associated with the event.
(price, φ, B)	the begin event of price element.
(price, {(text, "12:00")}, T)	the text event of price element. The text "12.00" is associated with the event.
(price, φ, E)	the end event of price element.
(name, φ, B)	the begin event of name element.
(name, {(text, "First")}, T)	the text event of name element. The text "First" is associated with the event.
(name, φ, E)	the end event of name element.
(author, φ, B)	the begin event of author element.

Table 2.1 : XML SAX events

at most one pair of the form (*a, v*) in the attribute list of any element, for all *a*.

- *type* is *B* for a begin event, *E* for an end event, and *T* for a text event. Events of type *E* have an empty attribute list, while events of type *T* have an attribute list containing the single pair (*text, t*), indicating that *t* is the text content of the element.

2.5 Summary

In this chapter we specified the difference between the three types of data models. We presented several important features of the XML data model together with some examples to explain their functionality. Also, we described the XPath query language and we focused on the fragment of XPath used in our implementation. In the next chapter we present some related systems, which have been developed to address the XML data processing problem.

Chapter 3

3 Related work

In the previous chapter, we discussed some important concepts of the XML data model and the XPath query language. This model and its language have been used in most of the systems for processing XML documents. In this chapter, we present the difference between the streaming and non-streaming XML data. Also, we discuss several systems which have been implemented to address the problem of filtering and querying XML data (streamed or non-streamed).

3.1 Streaming vs. non-streaming XML Data

The Extensible Markup Language (XML) [42] has become a well-established data format and an increasing amount of information is becoming available in XML form. The term streaming data is used to describe data items that are available for reading only once and that are provided in a fixed order determined by the data source. Applications that use such data cannot seek forward or backward in the stream and cannot revisit a data item seen earlier unless they buffer it on their own. Examples of data that occur naturally in streaming form include real-time news feeds, stock market data, sensor data, surveillance feeds, and data from network monitoring equipment. One reason for some data being available in only streaming form is that the data may have a limited lifetime of interest to most consumers. For example, articles on a topical news feed are not likely to retain their value for very long. Another reason is that the source of data may lack the resources required for providing non-streaming access to data. For example, a network router that provides real-time packet counts, error reports, and security violations is typically unable to fulfill the processing or storage requirements of providing non-streaming (so-called random) access to such data. Similar concerns may lead servers hosting large files to offer only streaming network access to data even though the data is available internally in non-streaming form. Finally, since sequential access to data is typically orders of magnitude faster than random access, it is often beneficial to use methods for streaming data on non-streaming data as well. In what follows, we focus on streaming data that is in XML form and use the term streaming XML to refer to XML data in all of the above scenarios.

3.2 Systems for Filtering Streaming XML Data

Several systems have addressed the problem of filtering a stream of XML documents [2, 13, 18, 28]. Those systems are most related to the XTrie system [9] which has been implemented in the dissertation. This problem has been referred to variously as selective dissemination of information (SDI), publish-subscribe (pub-sub), and query labeling. Briefly, filtering assumes that the input is a stream of documents that are to be matched with a given set of queries. A query is said to match a document if the result of evaluating the query on the document is non-empty. Since there is no output other than the identifiers of the documents matching each query, methods for filtering are simpler than those needed for querying. So, we may think of methods for filtering as starting points for the exploration of more general methods for querying. Filtering systems typically focus on supporting high throughput for a large number of queries using only a moderate amount of main memory.

The XFilter system [2] focuses on the problem of evaluating a large number of XPath filter expressions over every document in a stream of documents. It uses finite-state automata for each XPath expression. An important limitation of XFilter is that its space requirement can grow to a very large size as an input document is parsed, which can also increase the filtering time significantly.

The YFilter system [13] addresses a similar problem and uses one automaton to evaluate all submitted filter expressions. It combines all the automata into one big automaton that uses a run time stack to track all the possible states for all the queries. Instead of the index used by XFilter, YFilter uses query identifiers in the states to denote the queries corresponding to the results. Thus, YFilter shares processing among path expressions to eliminate redundant work. By indexing on substrings rather than individual element names, the XTrie index provides a much more effective indexing mechanism than XFilter and YFilter.

Automaton-based methods spend a significant amount of time matching transitions to incoming events; as a result, deterministic automata typically yield higher throughput than their nondeterministic [9, 2, 13] counterparts. However, as usual, the deterministic version of an automaton may require a large amount of memory. This problem is addressed in [18] by using a lazy deterministic finite state automaton. The main idea is to first build a naive finite-state automaton directly from the XPath expression. At run time, the system adds new states as needed on the since it does not need to use a stack to keep track of all possible states, its throughput is improved. Although the deterministic automaton requires more memory than its nondeterministic counterparts, an upper bound on the size of DFA is provided in [18]. The XTrie system [9] offers less throughput than [18]. On the other hand, XTrie has less space requirements than [18], which is an important issue for the system's scalability. Also [18] does not support predicates in XPath expressions.

The problem of query labeling is studied in [28]. The authors propose a requirements index as a dual to the traditional data index. A framework is provided to organize the index efficiently and to label the nodes in streaming XML documents with all the matched requirements in the index. The problem of validating XML streams using pushdown automata has been studied in [38]. (Briefly an XML document is said to be valid with respect to a given Document Type Definition (DTD), if the document structure obeys the grammar specified in the DTD [42].) This problem can also be considered as a filtering problem because the pushdown automaton can filter the documents that satisfy the DTD. The [38] ,like [18], requires a large amount of memory because the states in the automaton can grow in exponential manner.

3.3 Systems for Querying XML Data

The above systems, like XTrie, support filtering, not querying of XML streams. We use the term filtering to refer to the task of finding the documents (from a given set) that satisfy a given predicate and the term querying to mean the task of extracting relevant portions of data from one or more documents, or from streaming XML. As we said before, methods for filtering are simpler than those needed for querying. The querying of XML documents requires extra work. Thus, XTrie is more efficient than the systems for querying XML streams. This makes XTrie very attractive for publish/subscribe systems where the efficiency and scalability are very important issues. In this section we present systems for querying XML data, due to their relation with the XML processing problem.

3.3.1 Streaming XML Data

The XMLTK system [3] uses a lazy deterministic finite state automaton where new states are added as needed (at runtime) [18]. The determinism results in higher system throughput. The trade-off is that the deterministic automaton requires more memory than its nondeterministic counterpart. XMLTK supports XPath expressions that retrieve only parts of a document. However XMLTK does not support predicates in XPath expressions. Therefore whenever it encounters an element that matches the path expression in a query, it can write it directly to output and no buffering is needed. In contrast, if the query includes predicates, the membership of an element in the query result cannot be decided immediately in general.

The XSQ system [15] provides an efficient implementation of XPath for streaming XML data. It supports XPath queries that have multiple predicates, aggregations, closure axes, and output functions that permit extraction of portions of the stream. These features, especially in combination, complicate query processing. The implementation is based on a clean system design that centers on a hierarchical arrangement of pushdown transducers augmented with buffers and auxiliary stacks (HPDT). Furthermore, XSQ produces incremental results and buffers data in an optimal manner (least amount of data for the least amount of time possible). A notable feature of XSQ is that at any point during query processing, the data that is buffered by XSQ must necessarily be buffered by any streaming XPath query engine.

A transducer-based approach to evaluating XQuery queries on streaming data is presented in [31]. An XQuery is decomposed into sub expressions and each sub expression is mapped to an XML Stream Machine (XSM). Each XSM consumes the content of its input buffer and writes output to its output buffers. The output buffer of one XSM may be the input buffer of another. This producer-consumer relationship of XSMs through their buffers results in a network of XSMs. This network is merged into a single XSM that can be optimized if the DTD for the input data is available. In [35], a similar approach is used to evaluate regular path expressions with qualifiers over well-formed XML streams. That system proposes a transducer network model called SPEX, in which each transducer is generated from a regular path expression construct. The output tape of one transducer forms the input tape of another. XSM does not support XPath features such as aggregations, closures, and multiple predicates. The combined, optimized XSM is quite complicated, making it difficult to group similar queries.

An interesting feature of the XAOS system [5] for streaming XML is that it supports XPath's reverse axes, such as parent and ancestor. It uses two data structures called X-tree and X-dag to reduce the amount of streaming data buffered in a matching structure. Essentially, the X-tree is the parse tree of the XPath expression, with reverse axes permitted. The X-dag is the equivalent XPath representation with reverse axes removed. The X-dag is used as a pattern to filter the incoming stream to remove the irrelevant nodes. The relevant nodes are stored in the matching structure based on their relations in the X-tree. When the end of the stream is encountered, results are produced by traversing the matching structure. A drawback of this approach is that it does not output any results until the end of the stream is encountered. (For unbounded streams, a periodic evaluation of the matching structure could be used). Rewriting XPath queries with reverse axes into equivalent queries with only forward axes is studied in [35]. However, since the rewriting algorithm introduces node set comparison operations in the new expression, the approach is difficult to apply in a streaming environment. For example, for an expression

X[ancestor :: Y/Z], the rewriting algorithm produces X[/descendant :: Y[Z] /descendant :: node() = self :: node()].

3.3.2 Non-streaming XML Data

Several systems provide methods for querying non-streaming XML data. Galax [16] is a full-fledged XQuery query engine. It implements almost all of the XML Query Data Model along with the type system and dynamic semantics of the XML Query Algebra.

XQEngine [24] is a full-text search engine for XML documents that uses XQuery and XPath as its query language. XPath expressions and boolean combinations of keywords are used to query collections of XML documents. The engine creates a full-text index for every document before the document can be queried. It is difficult to adapt these systems for streaming data.

A topic closely related to XPath query processing is XML transformation. XSLT is a standard template-based language for transforming XML [47]. The popular implementation of XSLT in Saxon [25] is based on an in-memory materialization of the entire XML document and is therefore limited in the size of documents it can efficiently transform. By using a streaming XPath processor such as XTrie, we can design an XML transformation system that buffers only limited amount amounts of data.

The STX system [6] takes a different, more procedural, approach to transforming streaming XML. It uses templates to specify the operations that should be performed when data matching the template pattern is encountered. We may think of STX as a general-purpose event-driven programming environment that is not tailored to a specific query language. However, it may be used for XPath processing if we design a method for generating efficient STX templates from XPath queries. For example, if there are two predicates in an XPath query, we may create two variables in the program to store the current results of the predicates. When a predicate is evaluated, the corresponding variable is set to the result of the evaluation. We also need to specify explicitly when to reset the variables. We may then choose the right operation based on the current values of the variables. However, in this scheme, the positions of the elements have to satisfy the requirement that the predicate is evaluated before the target items. In general, it is not obvious how to generate STX templates equivalent to an XPath query in a systematic manner.

The query complexity of XPath is addressed by [17], which provides a main-memory algorithm for evaluating XPath on non-streaming data that is polynomial in the size of the query (and data). The method is based on reducing every axis to two primitive axes: first-child and next-sibling. The algorithm traverses the XPath parse tree in a bottom-up manner. The sub expressions in the lowest level are evaluated by scanning the data. The results of these sub expressions are then used in the evaluation of their parent sub expressions, recursively. The system also provides a refined top-down algorithm and suggests a core subset of XPath that can be evaluated in linear time. Since these methods require multiple passes of the data, it is not easy to adapt them methods for a streaming environment.

The evaluation of XPath queries over XML data is closely related to the problem of tree pattern matching [34, 11]. As described in [34], despite the resemblance, there are important differences between XPath evaluation and the classical problems of tree pattern matching [19] and unordered tree inclusion [26]. In particular, the problem of unordered tree inclusion is NP-hard (by direct reduction from SAT) [26], while XPath queries can be answered in polynomial time [17]. Intuitively, the reason the inclusion problem is harder than the XPath problem is that the former does not permit multiple nodes in the pattern tree to be mapped to the same node in the data tree. Most of the algorithms for these problems require a postorder (bottom-up) traversal of the data trees and are thus unsuitable for streaming data that is provided in preorder. As an exception, the algorithm described in [19] for the classical tree pattern matching problem, needs only a preorder traversal of the data tree. However, it allows only parent-child (not descendant) edges in patterns and finds only matches for which siblings occur in the same order in the data and as in the pattern. On the other hand, tree patterns corresponding to XPath queries include ancestor-descendant edges (for the closure axis) and XPath semantics require that the sibling order in the pattern (order of nodes mentioned in predicates) be ignored. Therefore, this algorithm cannot be easily applied to XPath.

An alternating automaton is an automaton in which each state has a indicating the acceptance or rejection [10]. There are three types of states: universal, existential, and negating. A universal (existential) state becomes an accepting state if all (respectively, at least one) of its offspring states reach accepting states. A negating state has a unique offspring and becomes an accepting state only when the offspring state is a rejecting state. There are two difficulties in applying alternating automata for streaming XPath evaluation: First, alternating automata naturally express the semantics of filtering expressions, but not querying expressions. In particular, they do not provide a mechanism to solve the address the buffering problems. Second, they use a bottom-up model of computation that does not fit well with the preorder arrival of streaming XML input. However, it may be possible to adapt some of the ideas used by alternating automata for XPath.

3.4 Other Systems

The Aurora system [7, 12, 49] is a data stream management system for monitoring applications, in which typical tasks include tracking the abnormalities among multiple streams, filtering specific target data for the user, and executing queries involving aggregations and joins. The Aurora system processes data streams using a large trigger network. The trigger, which is essentially a data-graph, is generated from the persistent queries provided by applications. The tuples in the results of these queries are created from the incoming streams and fed into the original application also in streaming form. The Aurora system provides a set of operators for an application to specify the persistent query and quality of service (QoS) requirements. At runtime, the Aurora system is optimized by using techniques such as load shedding (discarding data that requires a long time to process) and real-time scheduling.

The Fjords architecture [32] has been developed for managing multiple queries over the numerous data streams generated from sensors. Sensor data is generated in streaming form and the data rate is typically high and variable. The Fjords architecture is designed to maintain a high throughput for queries even when the data rate is unpredictable. It provides an efficient and adaptive infrastructure for more sophisticated query applications. The main components of the architecture are the queuing system and the sensor proxies. The queues can function in either pull or push mode. They are the basic functional structures to route data between the operators in a query plan. Query operators may be adaptive, such as Eddies [4]. Each sensor has a sensor proxy that accepts queries and tries to simplify the queries for the sensor's processor. The proxy adjusts the sample rate of the sensor based on the queries and permits different users share data from the sensor. Such optimizations result in higher throughput and longer sensor battery life, since energy is conserved by avoiding unnecessary sampling.

The NiagaraCQ system [11] is designed to efficiently support a large number of subscription queries expressed in XML-QL over distributed XML datasets. It groups queries based on their signatures. Essentially, queries that have similar query structure by different constants are grouped and share the results of the sub queries representing the overlap among the queries. Although NiagaraCQ handles both change-based and timer-based continuous queries, the events it handles (such as changed remote XML file and activated timer) are at a high level. Therefore, it can use materialized data that is managed by a cache manager. In contrast, the XTrie system responds to every event generated by a SAX-like parser. XTrie filters queries on streaming data, and the result is also in streaming form.

A related system, WebCQ, implements server-based Web page monitoring [29, 30]. Users use WebCQ's own query language to specify a sentinel, which is essentially a request for monitoring the specified Web objects. The sentinel supports different kinds of objects, such as images and links in Web pages, different time intervals for change detection, and different kinds of notification mechanisms. Although both WebCQ and XTrie are event-driven systems, the events in WebCQ systems are specified by the user and are mostly timer-based. When a timer is activated, WebCQ visits the specified Web resource and pulls the content that will be compared with its stored version in the cache.

Another system for processing data streams is dQUOB [36]. It views the data streams as a relational database. Each event in the stream maps to a tuple in a relation that characterizes the stream. It uses SQL [39] extended with create-if-then rules from Starburst's active database query language [48]. The create clause specifies the name of the rule and the data source, the if clause contains a SQL query, and the then clause specifies an optional function that accepts the result of the SQL query for further processing (including serving as the input of another query). The dQUOB system can generate optimized query plans for the continuous queries presented in the system based on the relational model and allows user-specified adaptation for changes in data streams.

Most work on streaming data assumes that the input consists of only the raw data. In this environment, certain limitations are unavoidable. For example, it is easy to devise XPath queries and sample inputs for which an unbounded amount of buffering is required for any XPath processor that produces exact results. An interesting alternative to this environment is one in which the input provides some assistance to the query processor by specifying constraints on forthcoming data or some other similar hints. For example, [40] describes a method for embedding punctuations in streaming data, facilitating the streaming evaluation of queries that include blocking operators such as group by. It should be interesting to use similar ideas for streaming XML to support XPath queries that include traversal axes.

3.5 Summary

In this chapter, we discussed some of the well-known systems related with the XML data processing problem. We distinguished the systems for filtering XML data streams, from those which query streaming and non-streaming XML data. Finally, we discussed some other related systems referenced to XML processing. In the next chapter, we will thoroughly present several important definitions about the key features which are used in the XTrie indexing model. We will discuss in detail the methodology for decomposing complex XPEs into substrings as described in [9] and we will show how these substrings can be organized to a rooted tree. Finally, we will present some important notions about the substring matchings [9] with XML documents.

Chapter 4

4 XPE Decompositions and Matchings

This chapter presents some important concepts that play key role in the XTrie indexing structure. We explain how an XPE can be represented as a rooted tree and match an XML document. Moreover, we describe the mechanisms for decomposing XPEs into sequences of XML element names (i.e., *substrings*), and explain how these substrings can be organized into substring-trees [9]. Then, we continue by discussing some important concepts for matching based on substring-trees. We close this chapter with a section about subtree-matching [9], where we describe how the subtrees can be used for effective matching over streaming XML documents.

4.1 XPE-Tree and XPE Matching

The XTrie index is dealing with a fragment of XPath known as tree patterns. A tree pattern expression is represented as a rooted tree. Each node is labeled with an element name prefixed by '/' or '//' and optionally by a sequence of one or more */. The ordering of nodes in the tree is determined by the order of appearance of their corresponding elements in the XML document examined. [9] refers to such a tree-structured representation of an XPE as an *XPE-tree*. As an example, Figure 4.1 depicts the XPE-tree of the expression *p* = *//a*[*//b*[**/c*]/*d*]]*/f*. Note that in Figure 4.1, the child node for **/c* precedes the child node for *d* since the former precedes the latter in the expression for *p*.

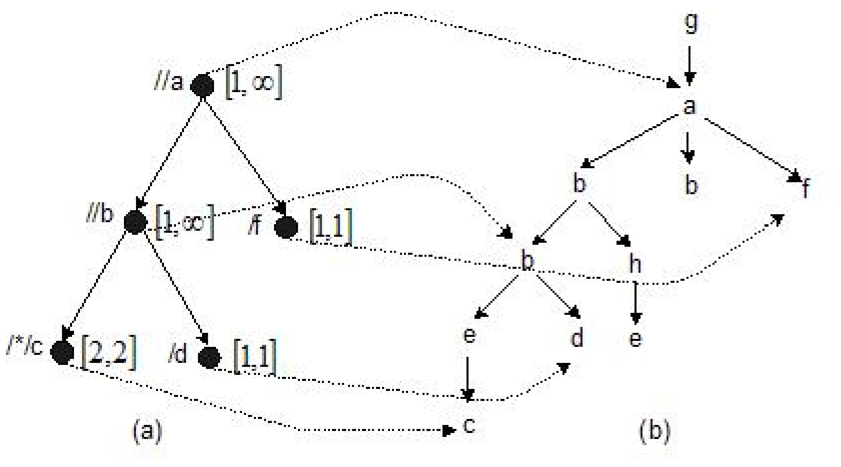


Figure 4.1 : XPath Tree matching with an XML Tree

***Definition*: [9]** *given two nodes υ and υ' in a rooted tree T, we say that υ precedes υ' in a pre-order traversal of T, denoted by $\upsilon <_{pre} \upsilon'$, if υ is visited before υ' in a pre-order traversal of T.*

The structure described leads to the definition of an interesting and useful metric, called the relative level [9] of a node. The relative level expresses the possible distance of a node from its closest ancestor in the XPE Tree.

***Definition*: [9]** *for each node t, if the label of t is prefixed with '//' followed by (k-1) '*/' then relLevel(t) = [k, ∞] (at least k). Otherwise, if t is prefixed by '/' followed by (k-1) '*/' its relative level is relLevel(t) = [k, k](exactly k).*

Also [9] defines the XPE-tree node mapping with an XML document node. Consider an XPE-tree *T* and an XML document tree *D*.

***Definition*:** [9] *a node t_i in T matches at a node d in D if the element name of t_i is equal to that of d.*

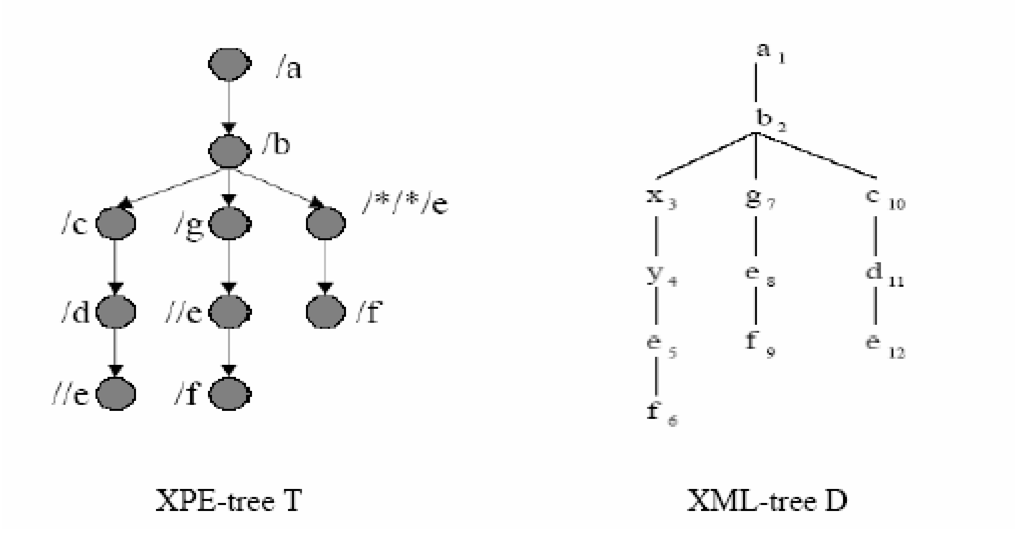


Figure 4.2 : An XPE Tree and an XML Tree example

Figure 4.1 shows the relative-level annotations for the nodes in our example XPE-tree and the node mapping indicated by the set of dashed arrows from the nodes in *T* to those in *D*.

4.2 Substring Decompositions

A *substring* [9] *s* of an XPath expression *p*, is a string produced by the concatenation of element names $s = t_1.t_2...t_n$ of nodes $u_1, u_2...u_n$ respectively, such that u_i is the parent node of u_{i+1} and every u_i (with the possible exception of u_1) is prefixed only by '/'. Let $P = <p_1, p_2, ..., p_n>$ a sequence of paths in the tree of p and let $<s_1, s_2, ..., s_n>$ be a sequence of strings where s_i derives from the concatenation of element names in p_i. If each s_i is a substring of *p* and:

1. Each node in the tree of *p* is in at least one p_i
2. p_i precedes p_j iff its last node precedes that of p_j in a pre-order traversal of XPE tree of *p*

then the sequence $<s_1, s_2, ..., s_n>$ is called a *substring decomposition* [9] of p. As an example, consider the XPE *p* = */a/b*[*c/d//e*][*g//e/f*]*// * / * /e/f* whose XPE-tree is depicted in Figure 4.2. The set of substrings of *p* includes *abg*, *bcd*, *ef* and *b*; on the other hand, *abge*, *gef*, and *bef* are not substrings of *p*, since they involve an intermediate element name (i.e., *e*) that is not prefixed by "/". There are two kinds of useful substring decompositions, *minimal* [9] and *simple* [9], which are used to enchase the filtering performance. In the next two sections we describe thoroughly these decompositions.

4.2.1 Minimal Decomposition

A *minimal decomposition* [9] *S of p* contains only substrings which have the maximal length (s_i ->max_{length}). In other words, there does not exist another longer substring s_j in *S* that contains s_i. Obviously a minimal decomposition of *p* is unique, since it comprise the smallest possible number of substrings among all possible decompositions of *p*. Figure 4.3a shows the minimal decomposition S_a for XPE *p*, where each dashed region encloses a path of nodes defining a substring containing in S_a. The choice of minimal decomposition in the XTrie index has two important advantages [9]:

- *Lower space*, due to fewer index probes since longer substrings (in minimal decomposition) have a lower probability being matched in the input XML document.
- *Best performance*, thus the cost of each index probe is generally lower with minimal decompositions (fewer XPEs associated with a longer substring).

On the other hand, the choice of minimal decomposition can result problems when checking for an ordered match of our XPE-tree with a streaming XML document. As an example, consider the minimal decomposition S_a = <*abcd, e ,abg, ef, ef*> of the XPE *p* and the XML document tree *D* in Figure 4.2, where the numeric subscripts denote the preorder traversal of the document elements through the SAX parsing interface. Clearly, *p* matches *D* in the unordered matching model. But in the ordering matching model (which is used in our implementation of XTrie), the matching of the substrings containing in the substring decomposition of *p* must has an order that enables the positional constraints between each matching substring and its "parent". For example, to correctly detect a matching of *ef*, the element *e* must be matched at exactly three levels below where the element *b* in *abcd* (or *abg*) is matched. The problem with this example is that the matching of *ef* (after f_6 is parsed in *D*) occurs

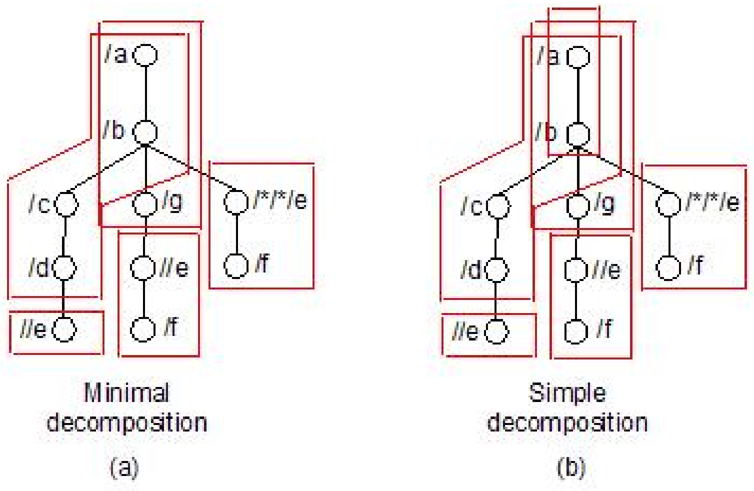


Figure 4.3 : Minimal and Simple XPE decompositions

before the matchings of both *abcd* and *abg* and, therefore, there is no matching occurrence of either of these substrings to enable checking the positional constraints for *ef*. This happens because, the substring *ab* appears only as a prefix of substrings *abcd* and *abg*, and not as an explicit substring in the decomposition of *p*. It is obvious, to avoid such problems, [9] uses a substring decomposition that enables the positional constraints in the XPE-tree. Thus, there is a need to enrich the minimal decomposition of an XPE so that it "takes note" of the branching nodes in the XPE-tree. This can be accomplished using another type of substring decomposition, the *simple XPE decomposition* [9].

4.2.2 Simple Decomposition

The simple decomposition *S* of an XPE *p* consists of two sequences S_1 and S_2 [9], where:

1. S_1 is the minimal decomposition of *p*.
2. S_2 consists of one substring *s* for each *branching node v* in *p*'s XPE-tree, such that *s* is the maximal substring in *p* with *v* as its last node and *s* is not already listed in S_1.

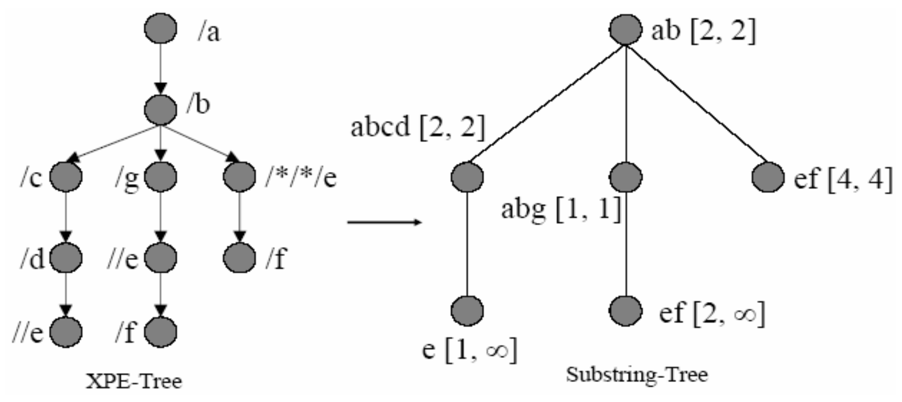


Figure 4.4 : Substring tree for S_b

Consider again the example for the XPE p in Figure 4.2, where the simple decomposition S_b of p depicted in Figure 4.3b. Note that S_b simply adds the substring *ab* (*b* is a branching node) to the minimal decomposition S_a. In addition, note that, for a single-path XPE (there isn't branching nodes), its simple decomposition is equal to its minimal decomposition.

4.3 Substring-Trees

Every substring in the substring decomposition (minimal, simple or other) of an XPE *p* can be organized into a unique rooted tree. This tree called *substring-tree* [9] of *p.*

Let $S = < s_1, s_2, \cdot\cdot\cdot, s_n >$ denote the simple decomposition of *p* corresponding to the sequence of paths $P = < p_1, p_2, \cdot\cdot\cdot, p_n >$ in the XPE-tree of *p* . Then, the substring-tree of *p* is constructed as follows [9]:

1. The *root substring* is s_1.
2. For each substring $s_i \in S$, $i > 1$, the *parent substring* of s_i is s_j (or equivalently, s_i is the *child substring* of s_j), if the last node of p_j (among all the paths in *P*) is the nearest ancestor node of the last node of p_i.
3. The ordering among sibling substrings is based on their ordering in *S*.

As an example, Figure 4.4 shows the substring-tree for the simple decomposition S_b of *p*. To continue, we must mention some important notions [9] of the substring-trees:

- *Substring Rank* [9]: the *rank* of s_i substring is k if s_i is the k^{th} child of its parent substring; the rank of the root substring is 1. For example, in Figure 4.4 the ranks of *abg* and *ef* (child of *abg*) are 2 and 1 respectively.
- *Leaf substring* [9]: a substring that has no child substrings. For example, in Figure 4.4, the substrings *e*, *ef*, and *ef* are *leaf substrings*.
- *Substring relative level* [9]: let V denote the set of nodes in p_i that are not in p_j. Let $x = \sum_{\upsilon_k \in V} l_k$, where *relLevel*(υ_k) = [l_k, u_k]. Then, the relative level of s_i denoted by *relLevel*(s_i) = [x, ∞] (a range) if $\max_{\upsilon_k \in V}\{u_k\} = \infty$. Else, denoted by *relLevel*(s_i) = [x, x] (exact value). Figure 4.4 shows the relative-level annotations for the nodes in the substring-tree.

4.4 Matching with Substrings

[9] extends the definition of matching for XPE nodes to substrings. Consider an XML document tree D and an XPE p with XPE tree T and simple decomposition $< s_1, s_2, \cdots, s_n >$ corresponding to the sequence of paths $P = < p_1, p_2, \cdots, p_n >$. Suppose p matches D; i.e., there is a node mapping f from the nodes in T to those in D. [9] defines the substring matching as follow:

***Definition*:** [9] *s_i matches at a node d in if f(υ) matches at d in D, where υ is the last node of p_i. We use $f(s_i) = \upsilon$ to denote a matching of s_i at node υ under the node mapping f. We say that there is a matching of s_i at level l in D if s_i matches at some node at level l in D.*

Clearly, to fully match p, [9] needs to find a matching for each of the substrings of p such that the positional constraint, defined by p between each substring and its parent, is satisfied.

As the nodes in D are parsed in a pre-order traversal (by the SAX parser [33]), the ordered matching of p in D also progresses incrementally following a pre-order traversal of the substring tree of p such that each substring s_i is matched before s_{i+1}. Thus, to determine if p matches D, [9] needs to keep track of the *partial matchings* [9] of p in D. However, since [9] is interested only in whether or not p matches D and not in the actual number of match occurrences, partial matchings of *p* that are *redundant* [9] should be ignored in order to improve the effectiveness of the filtering process. The next two sections define the notions of partial and redundant matchings.

4.4.1 Partial and Complete Matching

The *partial matching* [9] is a basic notion of the XTrie indexing scheme and can be defined as follow:

***Definition*:** [9] *there is a partial matching of substring s_i at a node d in D if the last node of p_i matches at d in D.*

[9] represent a partial matching by its node matching *f* that maps nodes from *T* to nodes in *D*.

***Definition*:** [9] *there is a complete matching of p in D if there is a partial matching of s_n the last substring in substring decomposition of p) at some node in D.*

4.4.2 Redundant Matching

Another important notion which play key role for the efficiency of XTrie filtering mechanism, is the *redundant matching* [9]. Its formally definition is:

***Definition*:** [9] *a partial matching of s_i at node d in D, where d is the k_{th} node in the pre-order traversal of D, is defined to be a redundant matching if for each XML document D' (that is equivalent to D for the first k nodes) that matches p under a mapping f with $f(s_i) = d$, there exists an alternative mapping f' that also defines a complete matching of p but with $f'(s_i) <_{pre} d$.*

As an example, consider the Figure 4.5, where the simple decomposition of *p* is < *a, b, c, bd* >. Note that the blue colored nodes in XML tree indicate the partial matchings of the substrings in the substring tree. In the other hand the red colored nodes indicate the redundant matchings. There are two redundant matchings:

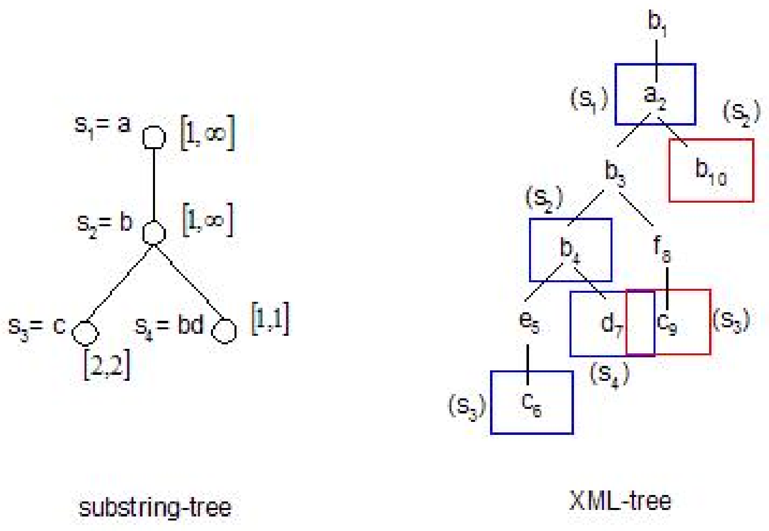


Figure 4.5 : Partial and redundant matching example

- (R1), the partial matching of substring *c* at the *c* node under the node *f*.
- (R2), the partial matching of substring *b* at the second *b* node under the node *a*.

Informally, a partial matching of a substring s_i is redundant if there already exists a preceding partial matching of s_i such that ignoring the later partial matching would not affect the correctness of deciding whether or not *p* matches *D* [9]. To achieve an efficient filtering (reducing the overhead of book-keeping operations) of documents with XPEs, [9] must detect and ignore the redundant substring matches. For that reason, [9] introduces the notion of *subtree-matchings* [9].

4.5 Subtree Matching

[9] extends the notion of *substring-matching* to the *subtrees* [9]. A formal definition of the *subtree-matching* is:

***Definition*:** [9] *a node mapping f is said to define a subtree-matching of s_i if f defines a partial matching of each descendant of s_i.*

Actually, *f* captures a matching that includes the entire XPE subtree rooted under s_i. As an example, consider again the substring tree in Figure 4.2, and assume that a

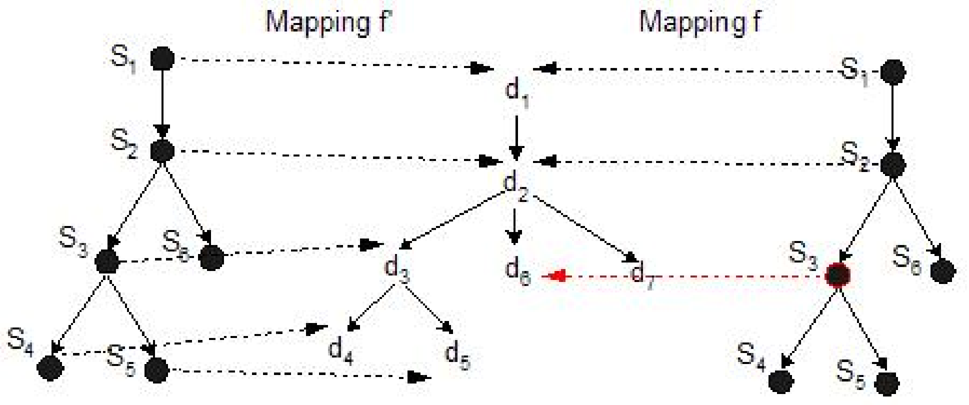


Figure 4.6 : Example of redundant matching

partial matching of the substring *ef* (whose parent substring is *abg*) has just been detected. This implies that there is a subtree-matching for each of the following four substrings: *abcd*, *abg*, *e* and *ef* itself. Also, referring to the two redundant matchings (R1) and (R2) in Figure 4.5, the partial matching of substring *c* in (R1) is redundant because there already exists a subtree-matching of its ancestor substring *b*, while the partial matching of substring *b* in (R2) is redundant because there already exists a subtree-matching of the substring *b* itself. Thus, [9] can detect redundant matchings by keeping track of subtree-matchings for the various substrings. Another, more formally definition of the redundant matching is:

***Definition*:** [9] *a partial matching of s_i (defined by a mapping f) is redundant if there exists another partial matching of s_i (defined by a mapping f') such that:*

1. *$F'(s_i) <$ pre $f(s_i)$*
2. *there exists an ancestor substring s_a of s_i such that:*
   - I. *$f'(s_a) = f(s_a)$*
   - II. *f' defines a subtree-matching of the child substring of s_a whose subtree contains s_i.*

To illustrate the above subtree-based conditions for redundant matchings, consider the example in Figure 4.6, where two node mappings, *f* and *f'*, are shown for matching a substring-tree with (six substrings) to an XML document (with seven nodes). Suppose that the node d_6 in the XML document has just been parsed and it matches the

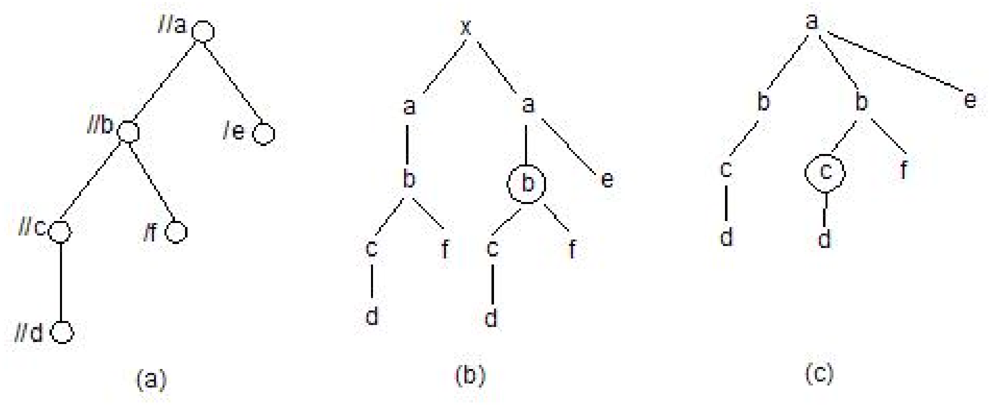


Figure 4.7 : Subtree-based conditions for redundant matchings

substring s_3. By the subtree-based conditions, the partial matching of s_3 at d_6 (defined by *f*) is redundant because there already exists an earlier partial matching of s_3 at d_3 (defined by *f'*) which is part of a subtree-matching (a subtree-matching of s_3 itself), where both *f'* and *f* map s_2, the parent substring of s_3, to the same node d_2. The XML document trees in Figure 4.7b illustrates why [9] needs the condition $f'(s_a) = f(s_a)$ in definition for redundant matchings. Without this condition on s_a, the partial matching of substring *b* to the circled *b* node would have been incorrectly considered to be redundant (it's not redundant because, ignoring the latter partial matching of substring b would not affect the correctness of deciding whether or not *p* matches *D)* since there is subtree-matching of substring *b* at an earlier *b* node. The XML document tree in Figure 4.7c illustrates why [9] needs the condition that there be a subtree-matching at the child substring of s_a (as opposed to at some *descendant* substring of s_a) whose subtree contains s_i. If the weaker condition is used, then the partial matching of substring *c* to the circled *c* node would have been incorrectly regarded as redundant (it's not redundant for the same reason as before), since there is a subtree-matching of substring *c* at an earlier *c* node and also, there exist an ancestor substring (*a*) of substring *c* such that $f'(a) = f(a)$.

4.6 Summary

In this chapter we presented some important definitions about the basic features which are used in XTrie. We explained how an XPE can be represented as a rooted tree and match an XML document. Moreover, we described the mechanisms for decomposing XPEs into substrings [9], and explained how these substrings can be organized into substring-trees [9]. Finally, we discussed some important concepts for matching based on substring-trees. In the next chapter we present in detail, the XTrie index structure and algorithms [9].

Chapter 5

5 XTrie Structure and Indexing Scheme

In this chapter, we present the XTrie indexing scheme for filtering XML documents based on XPEs. Moreover, we explain step by step all the algorithms in the XTrie scheme. Also, we explain how the XTrie mechanisms can deal with XPEs containing attributes and/or text data. We continue by presenting two maintenances of XTrie for XPE insertions [9] and deletions [9]. Finally, we discuss an optimized variant (Lazy) [9] of XTrie and explain in detail how the matching algorithms work.

5.1 The Index Structure

In this section we present two basic components of the XTrie index and explain their functionality. Let $P = \{p_1, p_2, \cdots, p_n\}$ denote the set of XPEs being indexed, and S denote the set of distinct substrings derived from all the simple decompositions of the XPEs in P. The XTrie index consists of two key components:

1. *Trie* [9] (denoted by T): constructed on S to facilitate detection of substring matchings in the input XML data.
2. *Substring-Table* [9] (denoted by ST): that stores information about each substring of each XPE in P. The information in ST is used to check for partial matchings.

Symbol	**Description**
P	Set of XPEs being indexed.
S	Set of distinct substrings from the simple decompositions of all the XPEs in P.
$\|p_i\|$	Number of substrings in the simple decomposition of p_i.
$s_{i,j}$	j^{th} substring in a decomposition of XPE p_i.
L_{max}	Maximum number of levels in XML document.
label(N)	Label of trie node N in XTrie.
$a(N)$	Substring pointer of trie node N in XTrie.
$\beta(N)$	Max-suffix pointer of trie node N in XTrie.

Table 5.1 : XTrie basic notations

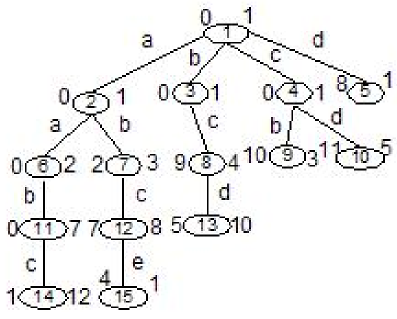
We now describe each of these two XTrie components in detail (using the notions in table 5.1), and present an example of XTrie index.

5.1.1 The Substring-Table

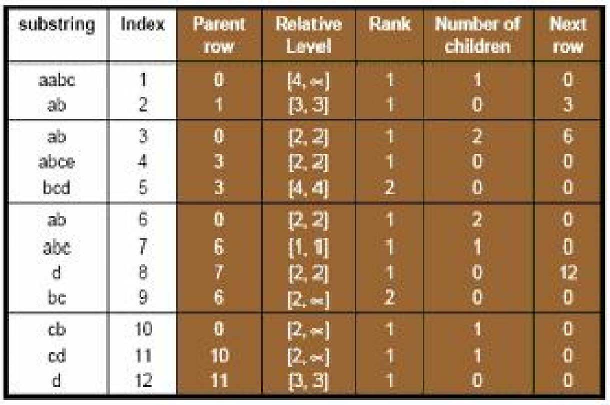
The substring-table *ST* contains one row for each substring of each indexed XPE; i.e., there are $\sum_{p \in P} |p|$ rows in *ST* with each row corresponding to some $s_{i,j}$ (denoting the j^{th} substring in the decomposition of p_i). The substrings derived from the same expression are stored in consecutives rows. [9] uses the symbol $r_{i,j}$ to denote the row in *ST* that corresponds to substring $s_{i,j}$. Finally, the rows representing the same substring are held into linked lists. Thus we have $|S|$ disjoint blocks in ST, one for each of these lists.

The information for each substring is held in one row. After examining the overall structure of ST, we focus to the structure of its rows. Each row is a tuple containing five fields:

- *ParentRow* [9]: an integer denoting the row in ST that contains the parent substring of this substring of $s_{i,j}$ (*ParentRow* = 0 if $s_{i,j}$ is a root substring).
- *RelLevel* [9]: obviously the relative level of $s_{i,j}$ (i.e., *relLevel*($s_{i,j}$)).
- *Rank* [9]: the rank of $s_{i,j}$ in the ordering of the children of its parent substring.
- *NumChild* [9]: is the total number of child substrings of $s_{i,j}$.



Trie



substring	Index	Parent row	Relative Level	Rank	Number of children	Next row
aabc	1	0	[4, ∞]	1	1	0
ab	2	1	[3, 3]	1	0	3
ab	3	0	[2, 2]	1	2	6
abce	4	3	[2, 2]	1	0	0
bcd	5	3	[4, 4]	2	0	0
ab	6	0	[2, 2]	1	2	0
abc	7	6	[1, 1]	1	1	0
d	8	7	[2, 2]	1	0	12
bc	9	6	[2, ∞]	2	0	0
cb	10	0	[2, ∞]	1	1	0
cd	11	10	[2, ∞]	1	1	0
d	12	11	[3, 3]	1	0	0

Substring Table

Figure 5.1 : An XTrie example

- *Next* [9]: is a "pointer" for a singly linked list, is the row number of the next tuple in *ST* that belongs to the same logical block as the current row. If the current row is the last row in the linked list, then *Next* = 0.

5.1.2 The Trie

The Trie [9] *T* is a rooted, node and edge-labeled tree. It is constructed from *S* (distinct substrings). Each edge is labeled with an element name, while each node *N* is associated with a label, *label*(*N*) [9]. That label is formed by the concatenation of every edge label that belongs in the path from the root of *T* to *N*. By this, [9] ensures that:

1. For each $s \in S$, there is a unique node *N* in *T* such that *label*(*N*) = *s*.
2. For each leaf node *N* in *T*, *label*(*N*) $\in$ ***S*.**

Each node in T has two special pointers apart of those to its children nodes:

- The *Substring pointer* [9] (denoted by *α*(*N*)): points to some row in *ST* (i.e., *α*(*N*) is a row number) determined as follows: if *label*(*N*) $\in$ *S*, then *α*(*N*) points to the first row of the linked list associated with substring *label*(*N*); otherwise, *α*(*N*) = 0.

- The *Max-suffix pointer* [9] (denoted by $\beta(N)$): points to some internal node in T and its purpose is to ensure the correctness of the matching algorithm. Specifically, $\beta(N) = N'$ if *label*(N') is the longest proper suffix of *label*(N) among all the internal nodes in T; if N' does not exist, then $\beta(N)$ points to the root node of T.

As an example, Figure 5.1 depicts the XTrie index structures for a set of four XPEs $P = \{p_1, p_2, p_3, p_4\}$ with their respective simple decompositions:

1. p_1 = *//a/a/b/c/*/a/b* *with* S_1 = <*aabc, ab*>
2. p_2 = */a/b[c/e]/*/b/c/d* *with* S_2 = <*ab, abce, bcd*>
3. p_3 = */a/b[c/*/d]//b/c* *with* S_3 = <*ab, abc, d, bc*>
4. p_4 = *//c/b//c/d/*/*/d* *with* S_4 = <*cb, cd, d*>

The number within each trie node N in Trie, as shown in Figure 5.1, represents the node's identifier, and the values of $\alpha(N)$ and $\beta(N)$ are shown to the left and right of N, respectively. Figure 5.1 also depicts the corresponding substring table with the rows clustered in the order of the XPEs in P.

5.2 The XTrie Matching Algorithm

The XTrie indexing scheme [9] is designed to support on-line filtering of streaming XML data and is based on the SAX [33] event-based interface that reports parsing events. Figure 5.3 depicts the search procedure for the XTrie, which accepts as input an XML document D and an XTrie index (ST, T), processes the parsing events generated by D, and returns the identifiers of all the matching XPEs in the index.

The algorithm accepts an XML document and an XTrie index (ST and T) to produce a set of identifiers indicating the matched XPath expressions. The basic steps are first to detect matching substrings using T and then examine the rows in ST that refer to this substring in order to check if this match is non-redundant. [9] needs some kind of dynamic information for the latter process, as ST contains only static information. For that purpose [9] introduces two more structures, arrays B [9] and C [9]. B and C are 2-dimensional arrays of size $|ST| \times L_{\max}$, where $L_{\max}$ is the maximum number of levels in the input document. In the next to sections we present those arrays and explain how they work.

5.2.1 B-array

The first array B [9] is an integer-array such that $B[r_{i,j}, l] = n$, $n > 0$, if there is a non-redundant matching of $s_{i,j}$ (represented by a node mapping f) at level l such that the n^{th} child substring of $s_{i,j}$ is the leftmost child substring of $s_{i,j}$ for which a subtree-matching has not yet been detected (i.e., f defines a subtree-matching of the $(n - 1)^{th}$ child substring of $s_{i,j}$). Intuitively, $B[r_{i,j}, l]$ records the rank of the *next child subtree of* $s_{i,j}$ *that we need to match* for this non-redundant occurrence of $s_{i,j}$ at level l. Thus, we know that an XPE p_i matches the input document when $B[r_{i,1}, l] = m + 1$ for some value of l, where m is the number of child substrings of the root substring $s_{i,1}$. Each $B[r_{i,j}, l]$ is initialized to 0, and is incremented to 1 after a non-redundant matching of $s_{i,j}$ at level l is detected. As more substring matchings are detected, the value of $B[r_{i,j}, l]$ is incremented from n to $n + 1$, $n \geq 1$, when there is a subtree-matching of the n_{th} child substring of $s_{i,j}$. The value of $B[r_{i,j}, l]$ is reset to 0 when the end-tag corresponding to the start-tag at level l is parsed.

5.2.2 C-array

The second array C [9] is a bit-array that is used to ensure that sibling substrings match along *distinct* branches for an ordered matching. Each entry $C[r_{i,j}, l]$ corresponds to a matching of the substring $s_{i,j}$ at level l, and is initialized with a value of 0. Whenever the value of $B[r_{i,j}, l]$ is incremented to some value $k > 1$, indicating that a subtree-matching of the $(k - 1)^{th}$ child substring of $s_{i,j}$ has been detected, $C[r_{i,j}, l]$ is set to 1. $C[r_{i,j}, l]$ is then reset back to 0 right before the next document node at level l is to be parsed (i.e., when an end-tag corresponding to a start-tag at level l is parsed in the input XML document). Informally, a value of $C[r_{i,j}, l] = 1$ indicates that the nodes parsed in the input document are along the same branch as the one that matched the $(k - 1)^{th}$ child substring of $s_{i,j}$; therefore, any matching of the k^{th} child substring of

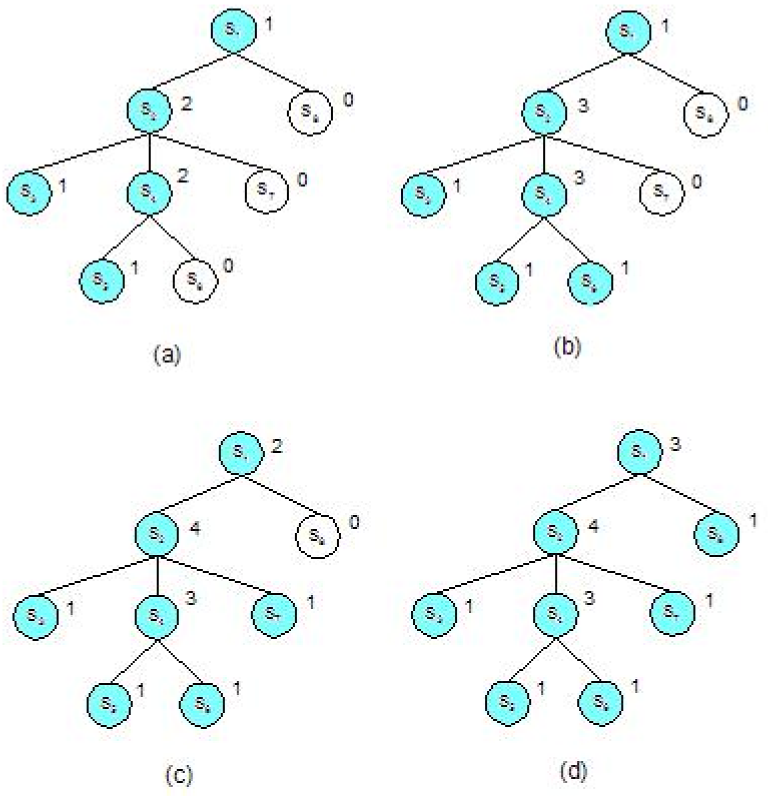


Figure 5.2 : Propagation of subtree-matchings example

$s_{i,j}$ (with $s_{i,j}$ matching at level l) detected during this period can not be considered a valid partial matching.

5.2.3 Detect non-redundant Matchings

To understand how the arrays B and C are used to detect non-redundant matchings, suppose that a matching of substring $s_{i,j}$ at level l has been detected, and $s_{i,j}$ is the n^{th} child substring of $s_{i,k}$. This matching is a partial matching of $s_{i,j}$ if there exists a matching of $s_{i,k}$ at level l' such that [9]: (1) $C[r_{i,k}, l']$ has a value of 0; (2) $l - l' \in relLevel(s_{i,j})$ (i.e., the positional constraint between $s_{i,j}$ and $s_{i,k}$ is satisfied); and (3) $B[r_{i,k}, l'] \geq n$ (i.e., we have subtree-matchings for at least the $n-1$ left-siblings of $s_{i,j}$ rooted at $s_{i,k}$). If, in addition, the value of $B[r_{i,k}, l']$ is exactly n, then this partial matching is non-redundant; otherwise, [9] has already discovered a subtree-matching for $s_{i,j}$, so the current matching is redundant and can safely be ignored. Note that since both B and C are large sparse arrays, their implementation can be optimized to

Start Tag	**Changes to *B* array after processing start tag**
g	
a	$B[a, 2] = 1$
b	$B[b, 3] = 1$
b	$B[b, 4] = 1$
e	
c	$B[c, 6] = 1$, $B[b, 4] = 2$
d	$B[bd, 5] = 1$, $B[b, 4] = 3$, $B[a, 2] = 2$, $B[b, 3] = 3$
h	
c	Redundant matching of *c* since $B[b, 3]$ is greater than the rank of *c*.
b	Redundant matching of *b* since $B[a, 2]$ is greater than the rank of *b*.
f	$B[af, 3] = 1$, $B[a, 2] = 3$, complete matching of *p*.

Table 5.2 : Execution trace of changes to *B* array for the matching of *p* on *D* in Figure 4.1

minimize space (e.g., using linked lists). In order to understand how the *B* array is used to detect non redundant matchings we present two concrete examples:

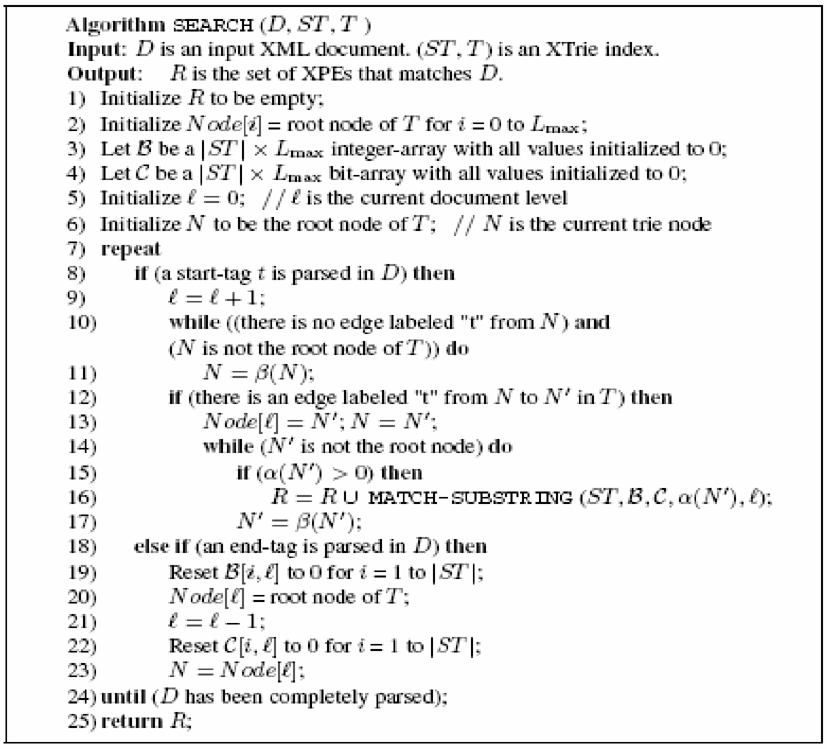
- *Example 1* [9]: consider the substring-subtree (consisting of substrings s_1 to s_8) in Figure 5.2a, which shows a partial matching of s_5. A shaded node for s_i means that there is a partial matching of s_i; and for notational convenience, assume that the partial matching of s_i ($1 \le i \le 6$) is at some node at level l_i of some XML document. The number to the right of each node s_i represents its $B[s_i, l_i]$ value. For instance, in Figure 5.2a, the *B* array value for s_2 is equal to 2 since only its first child substring (i.e., s_3) is part of a subtree-matching. Subsequently, when a partial matching of s_6 is detected, as shown in Figure 5.2b, it also trivially follows that there is a subtree-matching of s_6 since s_6 is a leaf substring. In order to correctly maintain the *B* array values, we need to propagate information about the subtree-matching of s_6 up to its parent substring (i.e., s_4) to indicate that a subtree-matching has been detected for its second child substring. This update propagation (indicated by an up arrow from s_6 to s_4 in Figure 5.2b) therefore increments s_4's *B* array value by one to 3, which in turn indicates that there is a subtree-matching of s_4. Consequently, [9] needs to further propagate the update upwards to s_2 and increment its *B* array value by one to 3. The update propagation stops at this point since there is no subtree-matching of s_2. Given the updated B array values in Figure 5.2b, it is clear that a subsequent partial matching of s_4 would be considered redundant since $B[s_2, l_2]$ is now greater than the rank of s_4. For a similar reason, a subsequent partial matching of either s_3, s_5, or s_6 is also considered redundant. Figures 5.2c and d, show how the B array values are updated after a partial matching of s_7 and s_8, respectively.

- *Example 2* [9]: Table 5.2 depicts an execution trace of the changes to the B array when matching XPE p against the XML document D in Figure 4.1. The second column of the table describes the changes to the B array after processing the start tag indicated in the first column. For instance, after the first c node in D is parsed, a partial matching of c, which is also a subtree-matching, is detected; and this is propagated to its parent substring b resulting in updates to both $B[c, 6]$ and $B[b, 4]$.

5.2.4 Details of Matching Algorithm

Eager XTrie uses three algorithms for matching XPEs over an XML document. These algorithms are:

- *SEARCH* [9] (depicted in Figure 5.3): begins by initializing the search node N to be the root node of the trie T (Step 6). For each start-tag t encountered, if there is an edge out of N with the label t (to another trie node N' in T), the search continues on node N'. For each trie node N' visited, a matching substring (corresponding to *label*(N')) is detected if $\alpha(N') \neq 0$; in this case, Algorithm *MATCH-SUBSTRING* [9] is invoked to process the matching substring using the substring table *ST*. Furthermore, for each trie node N' visited, algorithm also needs to check for other potential matching substrings that are suffixes of *label*(N'); this is achieved by using the max-suffix pointer (i.e., $\beta(N')$) in Step 17. On the other hand, if there is no edge out of a node N with the current tag t, this means that the concatenation of *label*(N) and t is not a matching substring. Therefore, algorithm needs to check for other potential matching substrings, which are formed by the concatenation of some suffix of *label*(N) and t, by using the max-suffix pointer in Step 11. For each end-tag t



```
Algorithm SEARCH ($D, ST, T$)
Input: $D$ is an input XML document. ($ST, T$) is an XTrie index.
Output:   $R$ is the set of XPEs that matches $D$.
1) Initialize $R$ to be empty;
2) Initialize $Node[i]$ = root node of $T$ for $i = 0$ to $L_{\max}$;
3) Let $B$ be a $|ST| \times L_{\max}$ integer-array with all values initialized to 0;
4) Let $C$ be a $|ST| \times L_{\max}$ bit-array with all values initialized to 0;
5) Initialize $\ell = 0$;  // $\ell$ is the current document level
6) Initialize $N$ to be the root node of $T$;  // $N$ is the current trie node
7) repeat
8)      if (a start-tag $t$ is parsed in $D$) then
9)          $\ell = \ell + 1$;
10)         while ((there is no edge labeled "t" from $N$) and
            ($N$ is not the root node of $T$)) do
11)             $N = \beta(N)$;
12)         if (there is an edge labeled "t" from $N$ to $N'$ in $T$) then
13)             $Node[\ell] = N'$; $N = N'$;
14)             while ($N'$ is not the root node) do
15)                 if ($\alpha(N') > 0$) then
16)                     $R = R \cup$ MATCH-SUBSTRING ($ST, B, C, \alpha(N'), \ell$);
17)                 $N' = \beta(N')$;
18)     else if (an end-tag is parsed in $D$) then
19)         Reset $B[i, \ell]$ to 0 for $i = 1$ to $|ST|$;
20)         $Node[\ell]$ = root node of $T$;
21)         $\ell = \ell - 1$;
22)         Reset $C[i, \ell]$ to 0 for $i = 1$ to $|ST|$;
23)         $N = Node[\ell]$;
24) until ($D$ has been completely parsed);
25) return $R$;
```

Figure 5.3 : Algorithm to search XTrie

encountered (corresponding to some start-tag at level *l*), the run-time information *B* is updated by resetting *B*[*r, l*] to 0 for all rows *r* (Step 19), and the search node is re-initialized to its previous location before the tag *t* was encountered (Step 20). This is achieved by using an array *Node* to keep track of the location of the search node at each document level (Step 13).

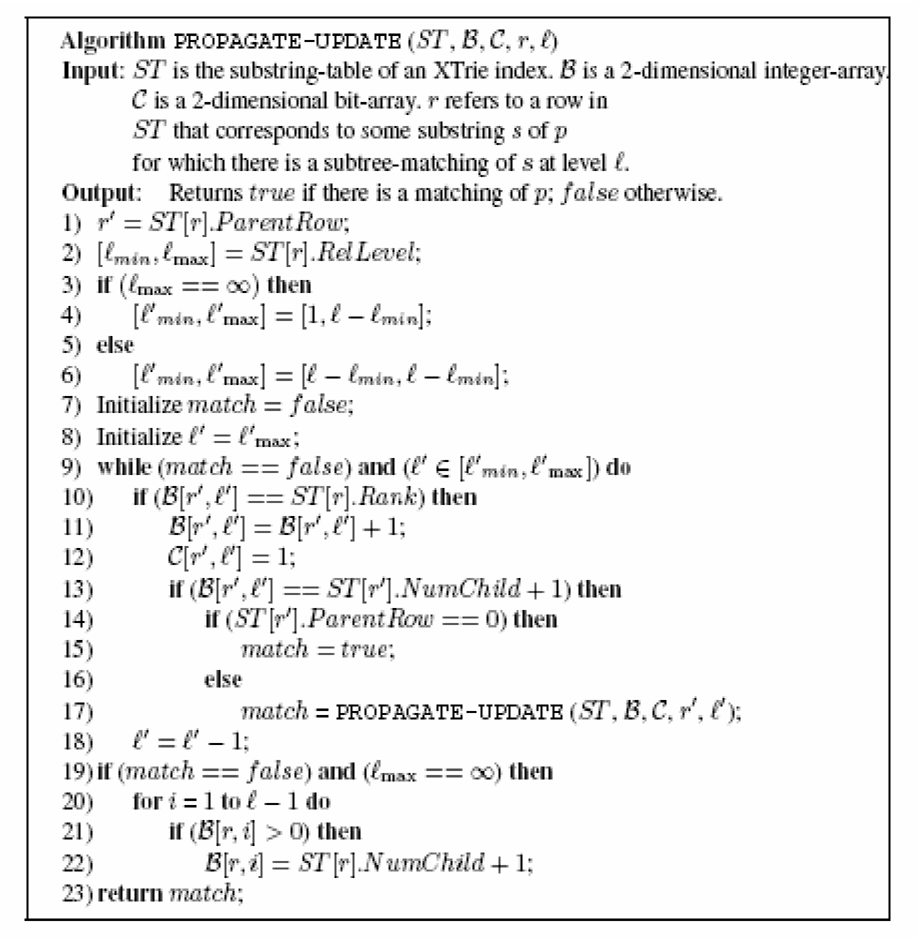
- *MATCH-SUBSTRING* [9] (shown in Figure 5.4): is invoked when a substring *s* (matching at level *l*) is detected. The algorithm checks for non-redundant matchings of *s*, updates the run-time information *B*, and returns the identifiers of all the matching XPEs that have *s* as their last substring. More specifically, the algorithm iterates through each instance of *s* in *ST* (i.e., each row in the linked list associated with *s*) to check for non-redundant matchings of *s*. There are two scenarios for the instance of the matching substring (say $s_{i,j}$) corresponding to row *r*. For the special case where $s_{i,j}$ is a *root substring* (Steps 5-9), if its positional constraint is satisfied (Step 6), then the matching is a partial matching (and obviously non-redundant, since it is a root substring), and *B*[*r, l*] is updated to 1 (to indicate that we can start looking for matchings

```
Algorithm MATCH-SUBSTRING ($ST, \mathcal{B}, \mathcal{C}, r, \ell$)
Input: $ST$ is the substring-table of an XTrie index. $\mathcal{B}$ is a 2-dimensional integer-array.
       $\mathcal{C}$ is a 2-dimensional bit-array.
       $r$ refers to the first row in $ST$ that corresponds to some substring
       that is matched at level $\ell$.
Output:   Set of matching XPEs.
1) Initialize $R$ to be empty;
2) while ($r \neq 0$) do
3)      $r' = ST[r].ParentRow$;
4)      Initalize $match = false$;
5)      if ($r' == 0$) then
6)          if ($\ell \in ST[r].RelLevel$) then
7)              $\mathcal{B}[r, \ell] = 1$;
8)              if ($ST[r].NumChild == 0$) then
9)                  $match = true$;
10)     else
11)         if ($\exists \ell' \in [1, \ell - 1]$ such that $\ell - \ell' \in ST[r].RelLevel$,
12)         $\mathcal{B}[r', \ell'] == ST[r].Rank$, and $\mathcal{C}[r', \ell'] == 0$) then
13)             $\mathcal{B}[r, \ell] = 1$;
14)             if ($ST[r].NumChild == 0$) then
15)                 $match$ = PROPAGATE-UPDATE ($ST, \mathcal{B}, \mathcal{C}, r, \ell$);
16)     if ($match$) then
17)         Insert the id. of the XPE corresponding to row $r$ into $R$;
18)     $r = ST[r].Next$;
19) return $R$;
```

Figure 5.4 : Algorithm to process a matched substring

of child subtrees). If, in addition, $s_{i,j}$ is a leaf substring, then we have a matching of p_i (Step 9). For the general case where $s_{i,j}$ is a *non-root substring* (Steps 10-15), if there is a non-redundant matching of $s_{i,j}$ (Step 11), then $B[r, l]$ is updated to 1. If, in addition, $s_{i,j}$ is a leaf substring, then Algorithm *PROPAGATE-UPDATE* [9] is called to update the run-time information arrays B and C, and check for a matching of the full XPE p_i. We should point out that, since [9] is not interested in finding multiple matches of the same XPE, [9] eliminates unnecessary processing and checking in *MATCH-SUBSTRING* for XPEs that have already been matched. This can be easily achieved by using a bit-mask (consisting of one bit per XPE).

- *PROPAGATE-UPDATE* [9] (depicted in Figure 5.5) is used to implement such “update propagations” and correctly update both B and C whenever a non-redundant subtree-matching of some non-root substring ($s_{i,j}$ matching at level l corresponding to row r in ST) is detected. The algorithm iterates through each matching of $s_{i,j}$ ’s parent substring (at level $l' \in [l'_{min}, l'_{\max}]$) and updates its B and C entries if the matching forms a non-redundant matching of $s_{i,j}$. If this



```
Algorithm PROPAGATE-UPDATE ($ST, \mathcal{B}, \mathcal{C}, r, \ell$)
Input: $ST$ is the substring-table of an XTrie index. $\mathcal{B}$ is a 2-dimensional integer-array.
        $\mathcal{C}$ is a 2-dimensional bit-array. $r$ refers to a row in
        $ST$ that corresponds to some substring $s$ of $p$
        for which there is a subtree-matching of $s$ at level $\ell$.
Output:   Returns $true$ if there is a matching of $p$; $false$ otherwise.
1)  $r' = ST[r].ParentRow$;
2)  $[\ell_{min}, \ell_{\max}] = ST[r].RelLevel$;
3)  if ($\ell_{\max} == \infty$) then
4)      $[\ell'_{min}, \ell'_{\max}] = [1, \ell - \ell_{min}]$;
5)  else
6)      $[\ell'_{min}, \ell'_{\max}] = [\ell - \ell_{min}, \ell - \ell_{min}]$;
7)  Initialize $match = false$;
8)  Initialize $\ell' = \ell'_{\max}$;
9)  while ($match == false$) and ($\ell' \in [\ell'_{min}, \ell'_{\max}]$) do
10)     if ($\mathcal{B}[r', \ell'] == ST[r].Rank$) then
11)         $\mathcal{B}[r', \ell'] = \mathcal{B}[r', \ell'] + 1$;
12)         $\mathcal{C}[r', \ell'] = 1$;
13)         if ($\mathcal{B}[r', \ell'] == ST[r'].NumChild + 1$) then
14)             if ($ST[r'].ParentRow == 0$) then
15)                 $match = true$;
16)             else
17)                 $match$ = PROPAGATE-UPDATE ($ST, \mathcal{B}, \mathcal{C}, r', \ell'$);
18)     $\ell' = \ell' - 1$;
19) if ($match == false$) and ($\ell_{\max} == \infty$) then
20)     for $i$ = 1 to $\ell - 1$ do
21)         if ($\mathcal{B}[r, i] > 0$) then
22)             $\mathcal{B}[r, i] = ST[r].NumChild + 1$;
23) return $match$;
```

Figure 5.5 : Algorithm to update run-time information arrays and detect complete matchings

matching is also a subtree-matching for the parent substring of $s_{i,j}$ (Step 13), then there are two cases to consider. If the parent substring is a root substring (Step 14), then [9] has found a matching of p_i; otherwise, [9] recurs the update propagation of the *B* and *C* entries for the ancestor substrings of $s_{i,j}$ as well (Step 17). The algorithm returns *true* if a matching of p_i has been detected; otherwise, if it is possible to have multiple matchings of the parent substring of $s_{i,j}$ (i.e., $relLevel(s_{i,j}) = [l_{min}, \infty]$ for some l_{min}), then, to avoid any subsequent redundant matchings of descendants of $s_{i,j}$, the algorithm updates the *B* entries of all the earlier matchings of $s_{i,j}$ (Steps 19 to 22), and returns *false*.

5.2.5 Theoretical Space and Time Complexity

The space requirement [9] of the XTrie index is dominated by the total number of substrings in P; that is, the space complexity is $O\left(\sum_{i=1}^{|P|} |p_i|\right)$, where $|p_i|$ denotes the number of substrings in the simple decomposition of p_i. To analyze the search-time complexity, let P denote the length of the longest root-to-leaf path in the trie T, let L denote the maximum length of a linked list in ST (i.e., the number of distinct occurrences of any substring), and let H denote the maximum height of a substring-tree [9]. The worst-case time complexity of Algorithm PROPAGATE-UPDATE is $O(H\ L_{\max})$ [9]. Since Algorithm MATCH-SUBSTRING makes at most L calls to Algorithm PROPAGATE-UPDATE, the complexity of Algorithm MATCH-SUBSTRING is $O(L\ H\ L_{\max})$ [9]. For each start-tag in the input document, Algorithm SEARCH makes at most P calls to Algorithm MATCH-SUBSTRING; thus, the worst-case complexity of processing each start-tag in an input document is $O(P\ L\ H\ L_{\max})$ [9]. Finally, it is easy to see that processing an end-tag takes $O(|ST|)$ time; thus, the overall (worst-case) time complexity of processing each tag in an input XML document is $O(\max\{P\ L\ H\ L_{\max}\ ,\ |ST|\})$ [9].

5.3 Attributes and Text Data

So far, our discussion of XTrie has been limited to XPEs that do not refer to any attributes or text data. In this section, we explain how XTrie can be easily extended to handle those features.

To handle XPEs with attributes, [9] needs to extend the substring-table ST with an additional column, *Attribute* [9], which is a pointer to a list of attributes (including any predicates) associated with the elements in a substring. For example, consider the XPE $p = /a[@name][@address]/b[@cost \leq 500]/c[d]$, where element a must have two attributes "*name*" and "*address*", and element b must have an attribute "*cost*" with a value of no more than 500. The simple decomposition of p consists of three substrings: $s_1 = ab$, $s_2 = abc$, and $s_3 = abd$. Let r_1, r_2, and r_3 denote the rows in ST that correspond to s_1, s_2, and s_3, respectively. Then, the *Attribute* value of row r_1 points to a linked list consisting of two entries with information about the attributes associated with the elements a and b. (Note that this information will not be repeated in rows r_2 and r_3 to avoid redundancy). In addition, since both elements *c* and *d* are not associated with any attributes, their values for *Attribute* is a null value representing an empty attribute list. By keeping track of the attributes (and their values if any) associated with the elements as they are parsed in an input XML document, the additional constraints on attributes can be easily verified for each matching substring. Thus, a matching for a substring *s* is considered to be a partial matching of *s* if all the attribute constraints associated with *s* are also satisfied.

Note that predicates that involved text values are handled in a similar manner as described for predicates involving attributes; where the substring-table is extended with an additional column, *Text* [9], which is a pointer to a list of predicates on the text values associated with the elements in a substring. Essentially, [9] uses the SAX parser to generate a single event for each start-element tag which consists of the element name, all the attributes specified in the start-element tag, and any text value enclosed after the start-element tag. In this way, any predicates associated with an element can be checked after its start-element event is reported by the SAX parser.

5.4 Maintenance

In this section, we present the maintenance algorithms for XTrie. First we present the maintenance of the max-suffix pointers in the trie *T.* Then we continue by presenting two maintenance algorithms, one for XPE insertions and one for XPE deletions.

5.4.1 Reverse Trie

One approach to efficiently maintain max-suffix pointers is to build an auxiliary suffix trie structure T_{rev} [9] on the set of reversed substrings so that for each node *N* in *T*, there exists an unique node *N'* in T_{rev} such that *label*(*N*) = *reverse*(*label*(*N'*)). By enhancing T_{rev} with special node pointers $\gamma(.)$ so that $\gamma(N')$ points to its associated node in *T* (i.e., $\gamma(N') = N$ iff *label*(*N*) = *reverse*(*label*(*N'*))), the max-suffix pointer value of a node *N* in *T* can be determined easily by traversing T_{rev} using *reverse*(*label*(*N*)): if *N'* is the last node reached by *reverse*(*label*(*N*)) in T_{rev}, and *N''* is the closest ancestor node of *N'* that has a non-null value for $\gamma(.)$, then $\beta(N)$ is given by $\gamma(N'')$.

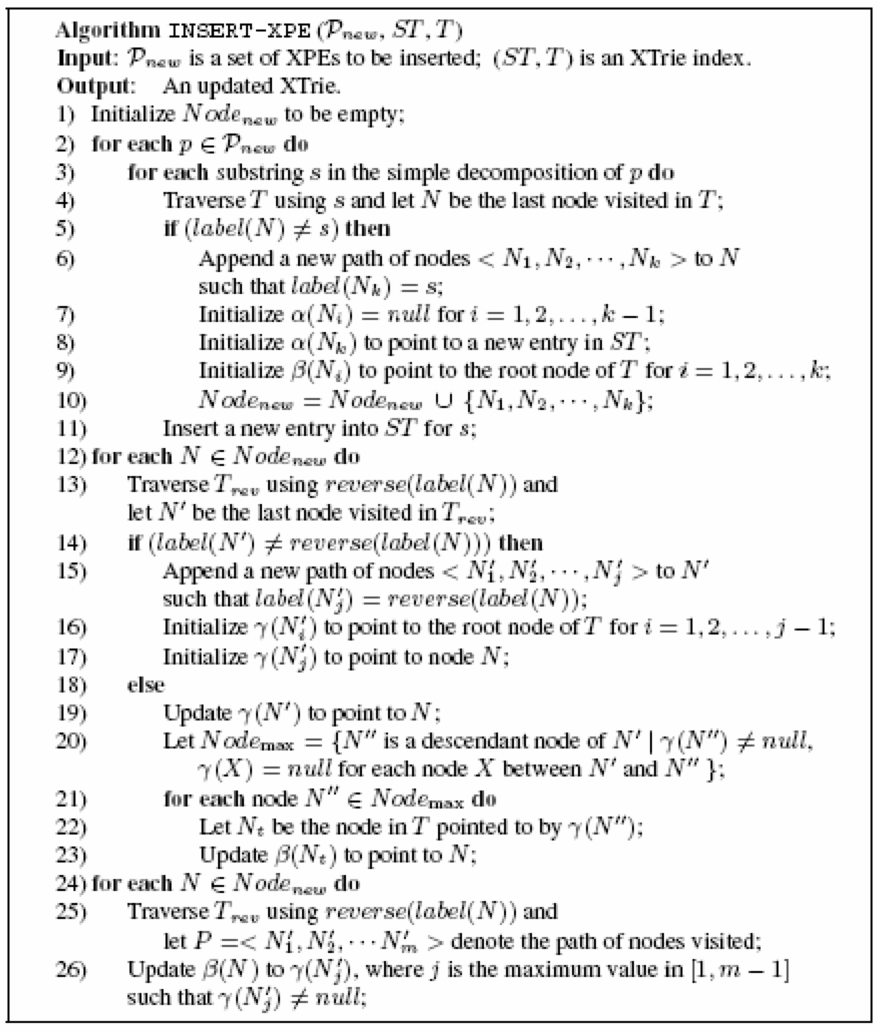
The auxiliary structure T_{rev} is basically a *suffix trie* on the set of reverse substrings in *S*; i.e., T_{rev} is a trie on the set $\{s' \mid s \in S, s' \text{ is a suffix of } reverse(s)\}$. The nodes in *T* and T_{rev} are related by the following invariant condition: for each node *N* in *T*, there exists a unique node *N'* in T_{rev} such that $label(N) = reverse(label(N'))$. [9] explicitly maintains this association between the nodes in *T* and T_{rev} by enhancing T_{rev} with additional pointers as follows: for each node *N'* in T_{rev}, [9] maintains a special pointer, denoted by $\gamma(N')$, that points to the node *N* in *T* (if it exists in *T*) such that $label(N) = reverse(label(N'))$; otherwise, $\gamma(N')$ is initialized to a null pointer value. Note that for the root node N'_{root} of *T'*, $\gamma(N'_{root})$ points to the root node of *T*. Given T_{rev}, the max-suffix pointer value of a node *N* in *T*, $\beta(N)$, can be easily computed as follows. Let $P =< N'_1 ,N'_2 , \cdots ,N'_k >$ denote the unique path of nodes in T_{rev} beginning from the root node N'_1 down to some node N'_k such that $label(N'_k) = reverse(label(N))$. Then, the value of $\beta(N)$ is given by $\gamma(N'_i)$, where N'_i $(1 \le i < k)$ is the bottom-most node in *P* (excluding node N'_k) that has a non null pointer value. Note that such a node always exists since $\gamma(N'_1)$ points to the root node of *T*.

5.4.2 Insertions

This section presents an algorithm [9] (shown in Figure 5.6) to update *XTrie* and the auxiliary structure T_{rev} when a new set of XPEs P_{new} is to be added. The maintenance algorithm to handle insertion of new XPEs consists of three main phases.

The first phase expands *T* with new nodes (if required) and updates *ST* with a new entry for each substring in the simple decomposition of each new XPE. The second phase expands T_{rev} with new nodes if new nodes have been inserted into *T* during the first phase, and updates their $\gamma(.)$ pointer values; this phase also updates the max-suffix pointers of some of the existing nodes in *T*. Finally, the third phase updates the max-suffix pointers of the new nodes that were added to *T* in the first phase. We now elaborate on the details of these three phases.

In the first phase (Steps 1 to 11), for each substring *s* in the simple decomposition of each new XPE, we traverse *T* using *s* to first check if there exists a node *N* in *T* such that $label(N) = s$. If not, an appropriate path of new nodes is inserted into *T* so that a leaf node in *T* is reachable using *s*. A new entry corresponding to *s* is also inserted into the substring table *ST*. The $\alpha(.)$ pointer values are updated



```
Algorithm INSERT-XPE ($\mathcal{P}_{new}$, $ST, T$)
Input: $\mathcal{P}_{new}$ is a set of XPEs to be inserted; $(ST, T)$ is an XTrie index.
Output:   An updated XTrie.
1) Initialize $Node_{new}$ to be empty;
2) for each $p \in \mathcal{P}_{new}$ do
3)      for each substring $s$ in the simple decomposition of $p$ do
4)          Traverse $T$ using $s$ and let $N$ be the last node visited in $T$;
5)          if $(label(N) \neq s)$ then
6)              Append a new path of nodes $< N_1, N_2, \cdots, N_k >$ to $N$
                such that $label(N_k) = s$;
7)              Initialize $\alpha(N_i) = null$ for $i = 1, 2, \ldots, k-1$;
8)              Initialize $\alpha(N_k)$ to point to a new entry in $ST$;
9)              Initialize $\beta(N_i)$ to point to the root node of $T$ for $i = 1, 2, \ldots, k$;
10)             $Node_{new} = Node_{new} \cup \{N_1, N_2, \cdots, N_k\}$;
11)         Insert a new entry into $ST$ for $s$;
12) for each $N \in Node_{new}$ do
13)     Traverse $T_{rev}$ using $reverse(label(N))$ and
        let $N'$ be the last node visited in $T_{rev}$;
14)     if $(label(N') \neq reverse(label(N)))$ then
15)         Append a new path of nodes $< N'_1, N'_2, \cdots, N'_j >$ to $N'$
            such that $label(N'_j) = reverse(label(N))$;
16)         Initialize $\gamma(N'_i)$ to point to the root node of $T$ for $i = 1, 2, \ldots, j-1$;
17)         Initialize $\gamma(N'_j)$ to point to node $N$;
18)     else
19)         Update $\gamma(N')$ to point to $N$;
20)         Let $Node_{max} = \{N''$ is a descendant node of $N' \mid \gamma(N'') \neq null$,
                $\gamma(X) = null$ for each node $X$ between $N'$ and $N''$ $\}$;
21)         for each node $N'' \in Node_{max}$ do
22)             Let $N_t$ be the node in $T$ pointed to by $\gamma(N'')$;
23)             Update $\beta(N_t)$ to point to $N$;
24) for each $N \in Node_{new}$ do
25)     Traverse $T_{rev}$ using $reverse(label(N))$ and
            let $P =< N'_1, N'_2, \cdots N'_m >$ denote the path of nodes visited;
26)     Update $\beta(N)$ to $\gamma(N'_j)$, where $j$ is the maximum value in $[1, m-1]$
        such that $\gamma(N'_j) \neq null$;
```

Figure 5.6 : Algorithm for insertion of a set of XPEs

appropriately, while the *β*(.) are simply initialized to point to the root node of *T* at this point.

In the second phase (Steps 12 to 23), [9] updates T_{rev} to maintain the invariant condition for the newly inserted nodes in *T*. Therefore, for each newly inserted node *N* in *T*, [9] traverses T_{rev} using *reverse*(*label*(*N*)) to check if there exists a node *N'* in T_{rev} such that *label*(*N*) = *reverse*(*label*(*N'*)). There are two possible cases. In the first case, if *N'* does not exist in T_{rev}, then an appropriate path of new nodes (with leaf node *N'*) is inserted into T_{rev} so that *label*(*N*) = *reverse*(*label*(*N'*)). The *γ*(.) pointer values for the newly inserted nodes in T_{rev} are updated appropriately. In the second case, if *N'* already exists in T_{rev}, then it is necessary that *γ*(*N'*) has a null pointer value (otherwise, it would imply that node *N* already exists in *T* contradicting the fact that *N* is a newly inserted node in *T*); thus, what remains to be done is to simply update *γ*(*N'*) to point to *N*. Since there might be some existing nodes in *T* whose max-suffix pointer values were initialized to *γ*(*N'*), we therefore need to update the max-suffix pointer values of such nodes to point to *N* instead. This update is performed in Steps 20 to 23. The set of nodes in T_{rev} associated with the affected nodes in *T* are represented by the set $Node_{\max}$; i.e., for each node $N'' \in Node_{\max}$, [9] needs to update the max-suffix pointer of the node pointed to by *γ*(*N''*). Note that the number of such affected nodes is bounded by the branching degree of node *N'* in T_{rev}.

Finally, the third phase (Steps 24 to 26), updates the max-suffix pointers for the newly inserted nodes in *T* as described in the previous section.

5.4.3 Deletions

This section presents an algorithm [9] (shown in Figure 5.7) to update *XTrie* when an existing XPE *p* is to be deleted. The maintenance algorithm consists of two main phases. The first phase deletes the appropriate entries in the *ST* that correspond to the substrings in the simple decomposition of *p*; nodes in *T* that have become "useless" as a result of the changes in *ST* are also deleted. The second phase deletes nodes in T_{rev} that have become "useless" and also updates those max-suffix pointers in *T* that are now pointing to non-existing nodes as a consequence of the nodes deleted in the first phase.

In the first phase (Steps 1 to 11), for each substring *s* in the simple decomposition of *p*, we delete the corresponding entry to *s* in *ST* by navigating to *T* via T_{rev}; that is we first traverse T_{rev} using *reverse*(*s*) to reach a node *N'* in T_{rev} and then navigate to its associated node *N* in *T* using *γ*(*N'*). The reason for this indirect navigation is because we need to "take note" of node *N'* in T_{rev} (by marking that node) if the node *N* in *T* is deleted. Note that a node *N* in *T* will be deleted if has become "useless"; i.e., *N* has become a leaf node and the value of *α*(*N*) has become a null pointer value. In order to efficiently ensure that all the useless nodes in *T* are deleted, [9] needs to visit these to-be-deleted nodes in a bottom-up manner; otherwise, [9] would have missed deleting an internal node that later becomes a useless leaf node. For this reason, [9] iterates through the to-be-deleted substrings in descending order

```
Algorithm DELETE-XPE ($p, ST, T$)
Input: $p$ is a XPE to be deleted; $(ST, T)$ is an XTrie index.
Output:   An updated XTrie.
1) Let $S_p$ be the set of distinct substrings from the simple decomposition of $p$;
2) Let $S_{sort}$ be the sorted sequence of substrings in $S_p$ in
   descending order of the substring length;
3) for each substring $s$ in $S_{sort}$ do
4)     Traverse $T_{rev}$ using $reverse(s)$ to reach node $N'$;
5)     Let $N$ be the node in $T$ pointed to by $\gamma(N')$;
6)     Access $ST$ using $\alpha(N)$ to delete the entry corresponding to $s$;
7)     if (the deleted entry is the last entry for $s$) then
8)         Update $\alpha(N)$ to the null pointer value;
9)         if ($N$ is a leaf node) then
10)            Delete $N$ from $T$;
11)            Mark the node $N'$ ;
12) for each substring $s$ in $S_{sort}$ do
13)     Traverse $T_{rev}$ using $reverse(s)$ to reach node $N'$;
14)     if ($N'$ is marked) then
15)         Let $N_{anc}$ be the closest ancestor node of $N'$ such that
            $N_{anc}$ is not marked and $\gamma(N_{anc}) \neq null$;
16)         Let $Node_{\max} = \{N''$ is a descendant node of $N' \mid \gamma(N'') \neq null$,
             N" is not marked,
               for each node $X$ between $N_{anc}$ and $N''$, $X$ is marked or $\gamma(X) = null\}$;
17)         for each node $N'' \in Node_{\max}$ do
18)             Let $N$ be the node in $T$ pointed to by $\gamma(N'')$;
19)             Update $\beta(N)$ to point to $\gamma(N_{anc})$;
20)         if ($N'$ is a leaf node) then
21)             Delete $N'$ from $T_{rev}$;
22)         else
23)             Update $\gamma(N')$ to a null pointer value;
```

Figure 5.7 : Algorithm for XPE deletion

of their lengths by first sorting them into the sequence S_{sort}. For each node in T that is deleted, its associated node in T_{rev} is marked for further processing in the second phase.

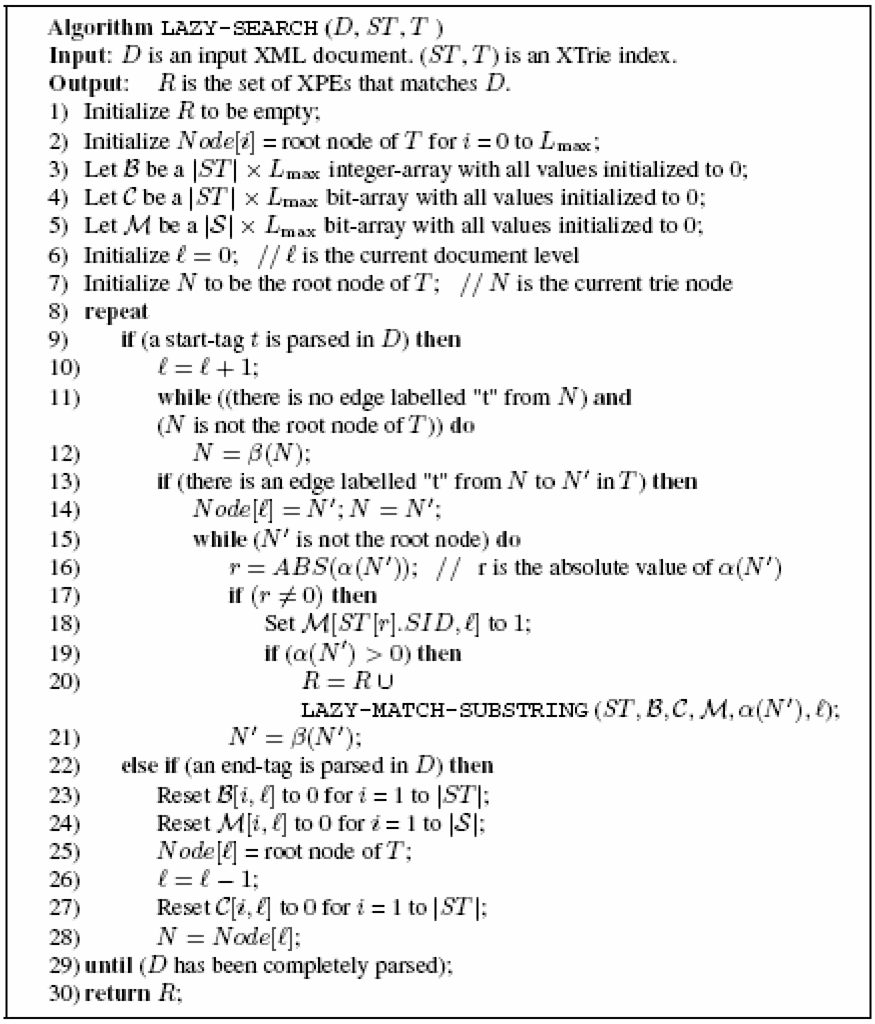
The second phase (Steps 12 to 23) begins once all the relevant entries in *ST* and useless nodes in T have been deleted. The purpose of this phase is to delete useless nodes in T_{rev} and update the max-suffix pointers in T using the updated T_{rev}. For each deleted node N in T, [9] first navigates to its associated marked node N' in T_{rev}. Node N' is deleted from T_{rev} if N' is a leaf node; otherwise, [9] updates $\gamma(N')$ to a null pointer value. The updating of the affected max-suffix pointers in T, which is performed in Steps 15 to 19, is similar to the procedure described earlier for the second phase in algorithm INSERT-XPE [9].

5.5 Lazy XTrie Optimization

In this section, we describe an optimization for XTrie. This optimization is based on a "lazy" XTrie variant [9] that aims to further reduce the number of unnecessary index probes. The XTrie variant that we have presented so far (referred to as *Eager XTrie*) probes the substring-table *ST* for every matching substring detected in the input document. The optimized *Lazy XTrie* variant tries to reduce the number of unnecessary index probes by postponing the probing of the substring-table *ST* so that *ST* is probed for a matching substring *s* only if *s* appears as a *leaf substring* in some XPE; otherwise, Lazy XTrie only updates information about the level at which *s* is matched in the input document. In this section, we explain the main differences between the lazy and eager variants of XTrie and present the matching algorithms used by Lazy XTrie.

An important consequence of this optimization is that the order in which substring matchings are processed in Lazy XTrie follows a bottom-up approach as opposed to Eager XTrie which follows the pre-order traversal of the XPEs substring-tree. To illustrate this difference, consider again the substring-tree in Figure 5.2. For Eager XTrie, the order of the partial matchings for the substrings follow the sequence $s_1, s_2, \cdots, s_8$. On the other hand, for Lazy XTrie, it first processes the matching of the leaf substring s_3 and then propagates upwards to process the matchings of substrings s_2 and s_1 (if they exist). Next, it detects and processes the matching of the second leaf substring s_5 followed by an upward propagation to process the matching of s_4 (if it exists). The remaining substrings (which are all leaf substrings) are detected and processed in the order s_6, s_7, and s_8. Thus, Lazy XTrie does not always immediately check if a matched substring constitutes a partial matching, but only does so in a bottom-up manner when the matched substring is a leaf substring. This difference in operation introduces a number of structural and algorithmic differences between Eager and Lazy XTrie.

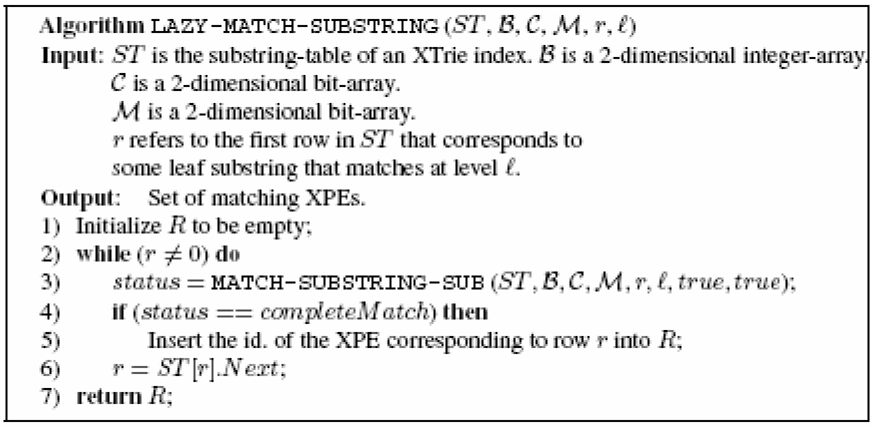
Structurally, Eager and Lazy XTrie are almost equivalent except for the following three differences. First, since Lazy XTrie only probes the substring-table when the matched substring *s* is some leaf substring, we need to "remember" all the matched substrings that have been detection prior to the matching of a leaf substring.



```
Algorithm LAZY-SEARCH ($D, ST, T$)
Input: $D$ is an input XML document. ($ST, T$) is an XTrie index.
Output:   $R$ is the set of XPEs that matches $D$.
1) Initialize $R$ to be empty;
2) Initialize $Node[i]$ = root node of $T$ for $i$ = 0 to $L_{\max}$;
3) Let $\mathcal{B}$ be a $|ST| \times L_{\max}$ integer-array with all values initialized to 0;
4) Let $\mathcal{C}$ be a $|ST| \times L_{\max}$ bit-array with all values initialized to 0;
5) Let $\mathcal{M}$ be a $|S| \times L_{\max}$ bit-array with all values initialized to 0;
6) Initialize $\ell = 0$;   // $\ell$ is the current document level
7) Initialize $N$ to be the root node of $T$;   // $N$ is the current trie node
8) repeat
9)      if (a start-tag $t$ is parsed in $D$) then
10)         $\ell = \ell + 1$;
11)         while ((there is no edge labelled "t" from $N$) and
            ($N$ is not the root node of $T$)) do
12)             $N = \beta(N)$;
13)         if (there is an edge labelled "t" from $N$ to $N'$ in $T$) then
14)             $Node[\ell] = N'$; $N = N'$;
15)             while ($N'$ is not the root node) do
16)                 $r = ABS(\alpha(N'))$;  //  r is the absolute value of $\alpha(N')$
17)                 if ($r \neq 0$) then
18)                     Set $\mathcal{M}[ST[r].SID, \ell]$ to 1;
19)                     if ($\alpha(N') > 0$) then
20)                         $R = R \cup$
                            LAZY-MATCH-SUBSTRING ($ST, \mathcal{B}, \mathcal{C}, \mathcal{M}, \alpha(N'), \ell$);
21)                 $N' = \beta(N')$;
22)     else if (an end-tag is parsed in $D$) then
23)         Reset $\mathcal{B}[i, \ell]$ to 0 for $i$ = 1 to $|ST|$;
24)         Reset $\mathcal{M}[i, \ell]$ to 0 for $i$ = 1 to $|S|$;
25)         $Node[\ell]$ = root node of $T$;
26)         $\ell = \ell - 1$;
27)         Reset $\mathcal{C}[i, \ell]$ to 0 for $i$ = 1 to $|ST|$;
28)         $N = Node[\ell]$;
29) until ($D$ has been completely parsed);
30) return $R$;
```

Figure 5.8 : Algorithm to search lazy XTrie

For this book-keeping, we maintain an additional data structure, denoted by *M*, which is a ($|S| \times L$max) bit-array such that *M*[*i, l*] is set to 1 if and only if the substring *s* is matched at level *l* of the input document, where $i \in [1, |S|]$ represents the identifier of *s*. For ease of access to the substring identifiers, [9] explicitly stores the substring identifiers in a new attribute, denoted by *SID* [9], in the substring-table such that $ST[r_{i,j}].SID$ is the identifier of the substring $s_{i,j}$. Second, in order to ensure that the substring-table is only probed for a matching leaf substring, [9] needs to distinguish between leaf and non leaf substrings. This is achieved by simply negating the values of $\alpha(N)$ in the trie if *label*(*N*) does not correspond to a leaf substring. Finally, unlike in Eager XTrie, where there are |*S*| linked lists in *ST* (with one list per distinct substring *S*); Lazy XTrie has only $|S_{leaf}|$ linked lists in *ST*, where $S_{leaf} = \{s \in S$ | s is a leaf



```
Algorithm LAZY-MATCH-SUBSTRING ($ST, \mathcal{B}, \mathcal{C}, \mathcal{M}, r, \ell$)
Input: $ST$ is the substring-table of an XTrie index. $\mathcal{B}$ is a 2-dimensional integer-array.
       $\mathcal{C}$ is a 2-dimensional bit-array.
       $\mathcal{M}$ is a 2-dimensional bit-array.
       $r$ refers to the first row in $ST$ that corresponds to
       some leaf substring that matches at level $\ell$.
Output:   Set of matching XPEs.
1)  Initialize $R$ to be empty;
2)  while ($r \neq 0$) do
3)      $status$ = MATCH-SUBSTRING-SUB ($ST, \mathcal{B}, \mathcal{C}, \mathcal{M}, r, \ell, true, true$);
4)      if ($status == completeMatch$) then
5)          Insert the id. of the XPE corresponding to row $r$ into $R$;
6)      $r = ST[r].Next$;
7)  return $R$;
```

Figure 5.9 : Algorithm to process a matching substring in lazy XTrie

substring in some XPE}; with one linked list for each substring in S_{leaf} such that a row $r_{i,j}$ in *ST* belongs to a linked list for substring *s* if and only if $s_{i,j}$ is a leaf substring of p_i and $s_{i,j} = s$. Thus, many of the rows in *ST* would not belong to any linked list at all.

Algorithmically, the main search algorithm [9] for Lazy XTrie is almost equivalent to that for Eager XTrie (in Figure 5.8) except that it now records occurrences of all matched substrings and probes the substring-table only when the matched substring is a leaf substring. However, checking if a matched substring *s* constitutes a partial matching in Lazy XTrie is more complex than in Eager XTrie due to the bottom-up approach of processing matched substrings in Lazy XTrie. In contrast to Eager XTrie, where the *B* array information about the ancestor substrings of a matched substring *s* have already been properly initialized to be used for processing *s*, this is not necessarily the case in Lazy XTrie. In particular, if *s* is the first child substring of its parent substring *s'*, then the *B* array information on *s'* has not been initialized and we first need to determine that there is a partial matching of *s'* itself, which might in turn lead to further propagation up the chain of ancestor substrings.

Moreover, Lazy XTrie uses the following algorithms for matching XPEs over an XML document. These algorithms are:

- *LAZY-MATCH-SUBSTRING* [9] (shown in Figure 5.9) is called to iterate through each instance of *s* in the indexed substrings via the linked list associated with *s* when a matching leaf substring *s* is detected; the input

```
Algorithm MATCH-SUBSTRING-SUB
($ST, \mathcal{B}, \mathcal{C}, \mathcal{M}, r, \ell, subpatternMatch, childSubpatternMatch$)
Output:   Returns one of the following status values:
          (1) $completeMatch$ if there is a matching of $p_i$,
          (2) $partialMatch$ if there is a partial matching of $s_{i,j}$ at level $\ell$, or
          (3) $noMatch$, otherwise.
1)  Initialize $status = noMatch$;
2)  $r' = ST[r].ParentRow$;
3)  if ($r' == 0$) then  //$r$ corresponds to a root substring
4)      if ($\ell \in ST[r].RelLevel$) then
5)          $status = partialMatch$;
6)  else  //$r$ corresponds to a non-root substring
7)      if ($subpatternMatch$ and ($ST[r'].NumChild == ST[r].Rank$)) then
8)          $parentSubpatternMatch = true$;
9)      else
10)         $parentSubpatternMatch = false$;
11)     $parentSid = ST[r'].SID$;
12)     Initialize $\ell' = \ell - \ell_{\min}$, where $ST[r].RelLevel = [\ell_{\min}, \ell_{\max}]$;
13)     while ($status \neq completeMatch$) and ($\ell' > 0$) and
        ($\ell - \ell' \in ST[r].RelLevel$) do
14)         if ($\mathcal{B}[r', \ell'] == ST[r].Rank$) and ($\mathcal{C}[r', \ell'] == 0$) then
15)             $status = partialMatch$;
16)             if ($parentSubpatternMatch$) then
17)                 $\mathcal{B}[r', \ell'] = ST[r'].NumChild + 1$;
18)                 if ($ST[r'].ParentRow == 0$) or
                    (PROPAGATE-UPDATE ($ST, \mathcal{B}, \mathcal{C}, r', \ell'$)) then
19)                     $status = completeMatch$;
20)             else if ($subpatternMatch$) then
21)                 $\mathcal{B}[r', \ell'] = \mathcal{B}[r', \ell'] + 1$;
22)                 $\mathcal{C}[r', \ell'] = 1$;
23)         else if ($\mathcal{M}[parentSid, \ell']$ and ($\mathcal{B}[r', \ell'] == 0$) and
            ($ST[r].Rank == 1$)) then
24)             $ret$ = MATCH-SUBSTRING-SUB
            ($ST, \mathcal{B}, \mathcal{C}, \mathcal{M}, r', \ell', parentSubpatternMatch, subpatternMatch$);
25)             if ($ret \neq noMatch$) then
26)                 $status = ret$;
27)         $\ell' = \ell' - 1$;
28) if ($status == partialMatch$) then
29)     if ($subpatternMatch$) then
30)         $\mathcal{B}[r, \ell] = ST[r].NumChild + 1$;
31)         if ($r' == 0$) then
32)             $status = completeMatch$;
33)     else
34)         if ($\mathcal{B}[r, \ell] == 0$) then
35)             $\mathcal{B}[r, \ell] = 1$;
36)         if ($childSubpatternMatch$) then
37)             $\mathcal{B}[r, \ell] = \mathcal{B}[r, \ell] + 1$;
38)             $\mathcal{C}[r, \ell] = 1$;
39) return $status$;
```

Figure 5.10 : Auxiliary algorithm to process a matching substring in lazy XTrie

parameter *r* refers to the first row in the substring-table that corresponds to *s*. For each matching substring $s_{i,j}$ ∈ *Sleaf* (matching at level *l* and corresponding to row *r* in *ST*), Algorithm *MATCH-SUBSTRING-SUB* [9] is invoked.

- *MATCH-SUBSTRING-SUB* [9] (shown in Figure 5.10) is invoked to check if this matching is a partial matching of $s_{i,j}$ and, if so, whether it also completes the matching of p_i. The algorithm returns one of the following three status values: *completeMatch* if there is a matching of p_i, *partialMatch* if there is a partial matching of $s_{i,j}$ at level l, or *noMatch* otherwise. The input parameter *subpatternMatch* is a Boolean variable indicating whether or not there is a matching of the subpattern rooted at $s_{i,j}$ (with $s_{i,j}$ matching at level l); and the input parameter *childSubpatternMatch* is a Boolean variable indicating whether or not there is a matching of the subpattern rooted at the most recently detected child substring of $s_{i,j}$. For the non-trivial case where $s_{i,j}$ is a non-root substring, the algorithm checks if the matching of $s_{i,j}$ at level l is a partial matching by iterating through each possible level l' for which the parent substring of $s_{i,j}$ (corresponding to row r' in *ST*) can be matched (i.e., $l - l' \in ST[r].RelLevel$) in Steps 13 to 27. There are three possible cases to consider. In the first case, if $B[r', l'] > ST[r].Rank$, then the matching is a redundant matching of $s_{i,j}$ and it can be ignored. In the second case, if $B[r', l'] = ST[r].Rank$, then the matching is a non-redundant matching of $s_{i,j}$; in addition, if the matching is also a subtree-matching of $s_{i,j}$ (Step 16) PROPAGATE-UPDATE (in Figure 5.5) is invoked to check if this leads to subtree-matchings of the ancestor substrings of $s_{i,j}$ and possibly a complete matching of p_i. In the third and final case, where $B[r', l'] < ST[r].Rank$, we have two possible sub-cases to consider. If $B[r', l'] > 0$, then there exists at least one preceding sibling substring of $s_{i,j}$ that has not been matched yet, which implies that the matching of $s_{i,j}$ is not a partial matching and can therefore be ignored. Otherwise, if $B[r', l'] = 0$, then in order for the matching of $s_{i,j}$ to be a partial matching, it is necessary that there is a partial matching of the parent substring of $s_{i,j}$ at level l' and $s_{i,j}$ is its first child substring. Therefore, a recursive call to Algorithm MATCH-SUBSTRING-SUB is made in Step 24 to check if there is a partial matching of its parent substring at level l'. Depending on the status of the matching of $s_{i,j}$, its B entry is updated accordingly in Steps 28 to 38.

5.6 Summary

In this chapter, we presented the XTrie indexing scheme for filtering XML documents based on XPEs. Also, we explained in detail all the algorithms which are used by

XTrie. We explained how the XTrie can deal with XPEs containing attributes and/or text data. We continued by presenting two maintenances of XTrie, one for XPE insertions [9], and one for XPE deletions [9]. Finally, we presented an optimized variant (Lazy) [9] of XTrie and the differences from the eager variant. In the next chapter, we present the XTrie architecture and implementation, giving the basic data structures and their attributes.

Chapter 6

6 System Architecture and Implementation

In Chapter 4 and 5, we gave in detail the basic concepts of the XTrie system and we presented several important definitions. Also, we explained how the matching algorithm works. In this chapter, we present our XTrie implementation and the basic structure of every part of the system's architecture.

6.1 XTrie Architecture

The XTrie system consists of 6 separated parts as shown in Figure 6.1. Each one has a specific functionality. Every part takes one input and returns a modified output. The output of each part is the input of the next part in the architecture. These parts are:

- *XPath parser*: It takes as an input an XPath expression and parses it into a sequence of location steps. Every location step has a tag, operator and a sequence of zero or more filters (attributes, text or other XPath expression).
- *XPE tree constructor*: It takes the location steps produced by the XPath parser and constructs the XPE tree. Also, it finds the relative level [9] of every node in the XPE tree.
- *Substring tree constructor*: It takes the XPE tree and finds the simple decomposition [9] of the XPE. Then it constructs the substring tree [9] using the simple decomposition. Further, it finds the relative level [9], rank [9],

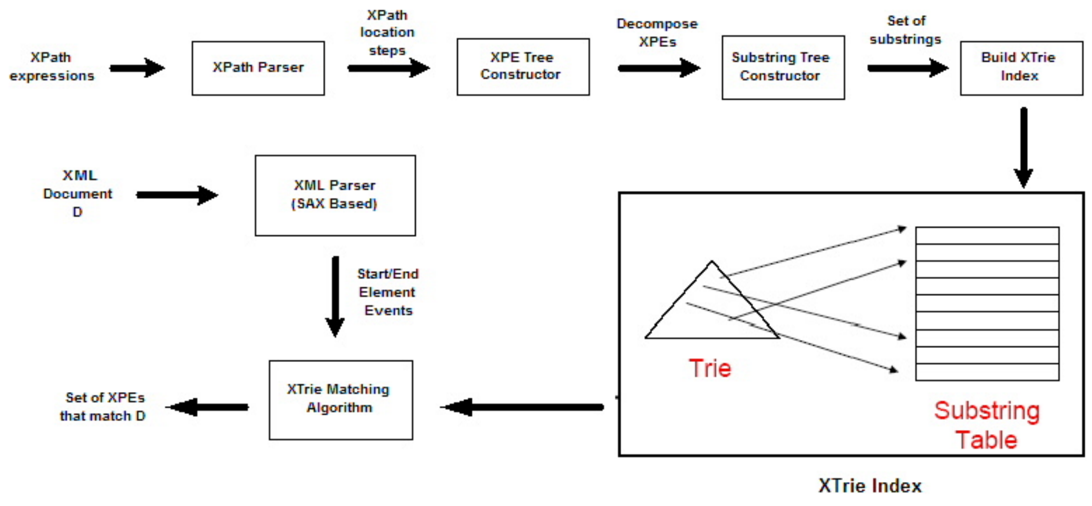


Figure 6.1 : The XTrie Architecture

number of children [9] and the parent [9] of every substring in the substring tree.

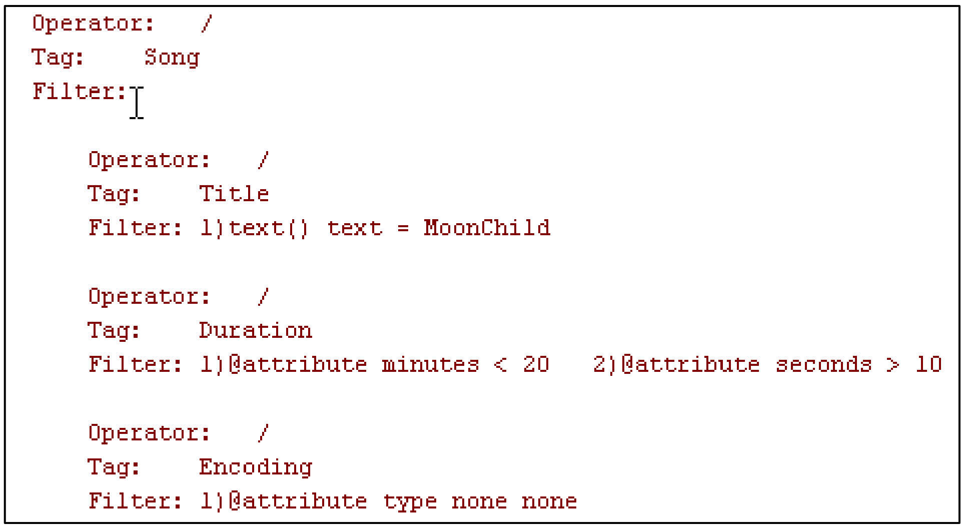
- *XTrie Index builder*: It constructs the Trie [9] and the Substring Table [9] using the substring tree.
- *XML parser (SAX based)*: It parses an arriving XML document and generates parsing events when it sees a *start* tag, an *and* tag and *data* internal of an element node.
- *Matching algorithm*: It takes as an input the XTrie structure and the SAX-based event produced by the XML document parser and returns the identifiers of the matching XPEs with the XML document.

6.2 XTrie Implementation

We have implemented the XTrie system in Java [23] using the Sun Java SDK version 1.4 [22]. Now we will describe in detail the structure and the methods of every part in the XTrie architecture.

6.2.1 XPath Parser

The XPath parser takes an XPE and produces several location steps. Each step consists of a *tag*, an *operator* and zero or more *filters*. The filter is in form of *id name*



```
Operator:    /
Tag:     Song
Filter:

    Operator:    /
    Tag:     Title
    Filter: 1)text() text = MoonChild

    Operator:    /
    Tag:     Duration
    Filter: 1)@attribute minutes < 20   2)@attribute seconds > 10

    Operator:    /
    Tag:     Encoding
    Filter: 1)@attribute type none none
```

Figure 6.2 : Location steps produced by the XPath parser

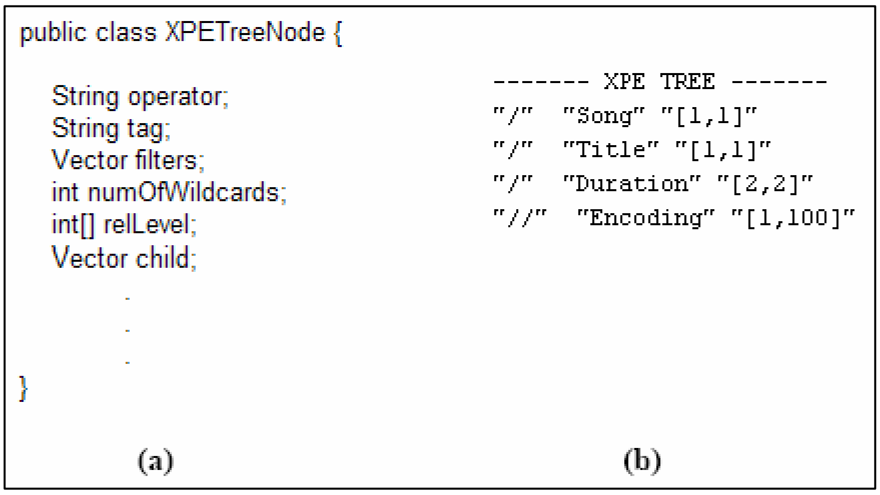
op const in case we have *attribute* or *text* comparisons. Also, a filter could contain a *relative XPE* to the context node. Thus, we have three kinds of filters:

- *Text*: where is in form of *text() text op const.*
- *Attribute*: where is in form of *@attribute attribute-name op const*. Note that, in this case, the op and const are optional.
- *Relative XPE*: where is in form of *tag operator *filter*.

As an example, consider the following XPath expression:

/Song[Title[text()='MoonChild']/Duration[@minutes<20][@seconds>10]/Encoding[@type]]

Our XPath parser takes this XPE as input and produces the location steps in Figure 6.2. The */Song [Title [text()='MoonChild'] /Duration[@minutes<20] [@seconds>10] /Encoding [@type]]* location step has a filter with the relative path *Title[text()='MoonChild'] /Duration [@minutes<20] [@seconds>10] /Encoding[@type]*. The */Title[text()='MoonChild']* location step has a text filter with value *'MoonChild'*. The */Duration [@minutes<20] [@seconds>10]* location step has two attribute filters. The first has an attribute name *minutes* with value smaller than *20* and the second filter has an attribute name *seconds* with a value greater than *10*. Finally the location



```
public class XPETreeNode {

    String operator;
    String tag;
    Vector filters;
    int numOfWildcards;
    int[] relLevel;
    Vector child;
            .
            .
            .
}
```

(a)

```
------- XPE TREE -------
"/"  "Song" "[1,1]"
"/"  "Title" "[1,1]"
"/"  "Duration" "[2,2]"
"//"  "Encoding" "[1,100]"
```

(b)

Figure 6.3 : (a) XPE Tree node structure and (b) XPE Tree

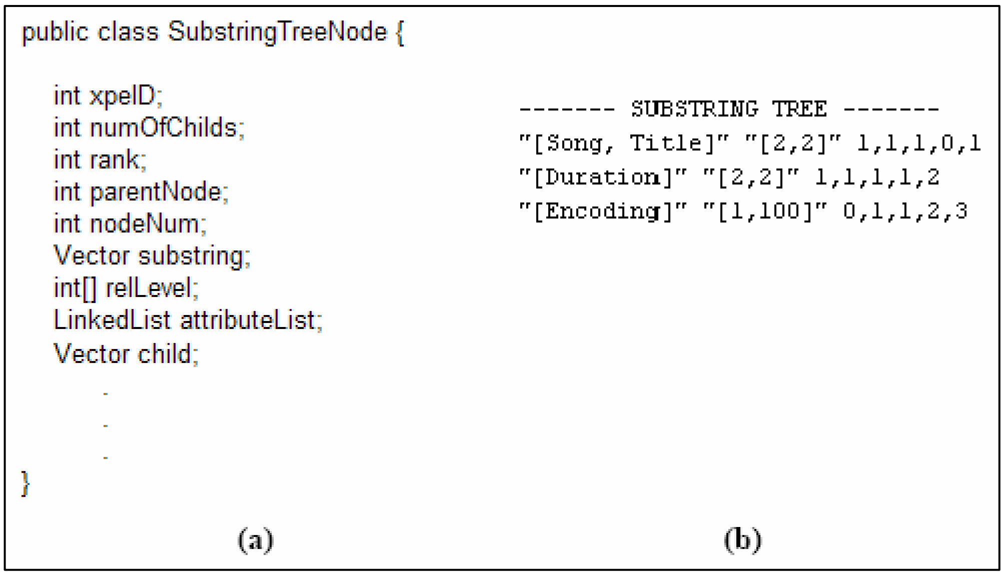
step */Encoding[@type]* has an attribute filter with attribute name *type*. Note that, in this case, there isn't any attribute comparison with some value.

6.2.2 XPE Tree Constructor

The XPE tree constructor takes the location steps from the XPath parser and constructs the XPE tree. The structure of the XPE tree node is depicted in Figure 6.3a. The XPE tree node has 6 fields:

- *operator*: is the operator of the location step (/ or //).
- *tag*: is the element name of the location step.
- *filters*: is a Vector which contains all the text and attribute filters of the location step.
- *numOfWildcards*: is an integer with the number of wildcard operators that prefix the operator of the location step.
- *relLevel[ ]*: is the relative level of node.
- *child*: is a pointer to a Vector of the node's children.

The XPE tree constructor takes the location steps in Figure 6.2, produced by the XPath parser, and constructs the XPE tree in Figure 6.3b.



```
public class SubstringTreeNode {

    int xpeID;
    int numOfChilds;
    int rank;
    int parentNode;
    int nodeNum;
    Vector substring;
    int[] relLevel;
    LinkedList attributeList;
    Vector child;
        .
        .
        .
}
```

(a)

```
------- SUBSTRING TREE -------
"[Song, Title]" "[2,2]" 1,1,1,0,1
"[Duration]" "[2,2]" 1,1,1,1,2
"[Encoding]" "[1,100]" 0,1,1,2,3
```

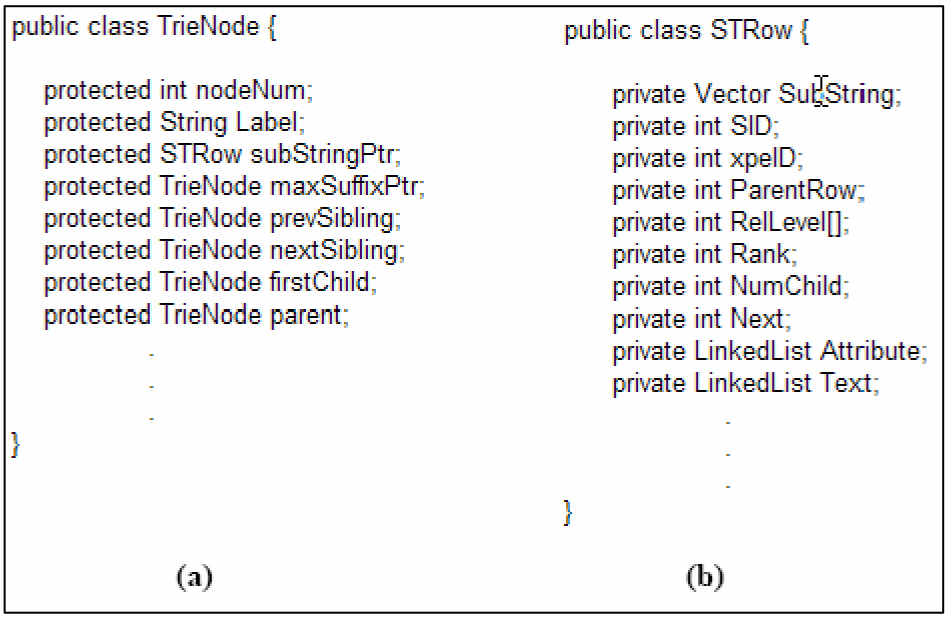
(b)

Figure 6.4 : (a) Substring Tree node structure and (b) Substring Tree

6.2.3 Substring Tree Constructor

The substring tree constructor takes the XPE tree and creates the simple substring decomposition of the XPE and constructs the substring tree. Figure 6.4a, depicts the fields of the substring tree node:

- *xpeID*: is an integer with the identifier of the XPE.
- *numOfChilds*: is an integer with the number of substring children.
- *rank*: is an integer with the position of the substring node on its fathers Vector child.
- *parentNode*: is an integer with the node number of its father substring.
- *nodeNum*: is the number of the node.
- *Substring*: is a Vector which contains the substring name of the node.
- *relLevel*: is the relative level of the substring node.
- *attributeList*: is a LinkedList which contains the attribute and text filters of every substring node (e.g. contains all the filters of every element in the substring).
- *child*: is a pointer to a Vector containing the children of the substring node.



```
public class TrieNode {

    protected int nodeNum;
    protected String Label;
    protected STRow subStringPtr;
    protected TrieNode maxSuffixPtr;
    protected TrieNode prevSibling;
    protected TrieNode nextSibling;
    protected TrieNode firstChild;
    protected TrieNode parent;
                -
                -
                -
}

                    (a)
```

```
public class STRow {

    private Vector SubString;
    private int SID;
    private int xpeID;
    private int ParentRow;
    private int RelLevel[];
    private int Rank;
    private int NumChild;
    private int Next;
    private LinkedList Attribute;
    private LinkedList Text;
                -
                -
                -
}

                    (b)
```

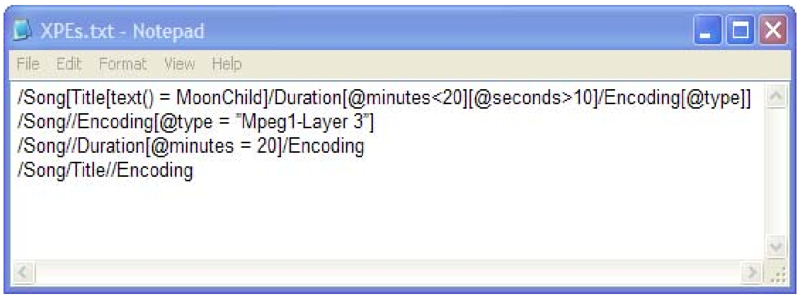
Figure 6.5 : (a) Trie node structure and (b) ST Row structure

The substring tree constructor takes the XPE tree, as depicted in Figure 6.3b, in order to find the substring decomposition and constructs the substring tree, as shown in Figure 6.4b. The numbers next to the substring relative level are: number of children, rank, XPE identifier, parent node number and node number respectively.

6.2.4 XTrie Builder

The XTrie builder consists of two components: a) the Trie constructor and b) the Substring Table constructor. The Trie is a rooted tree. Figure 6.5a, shows the fields of the Trie node:

- *nodeNum*: is the number of the node.
- *Label*: is the element name of the node.
- *subStringPtr*: is a pointer to some row in the substring table.
- *maxSuffixPtr*: is a pointer to an internal node.
- *prevSibling*: is a pointer to the previous node.
- *nextSibling*: is a pointer to the next node.
- *firstChild*: is a pointer to the child node.
- *Parent*: is a pointer to the father node.



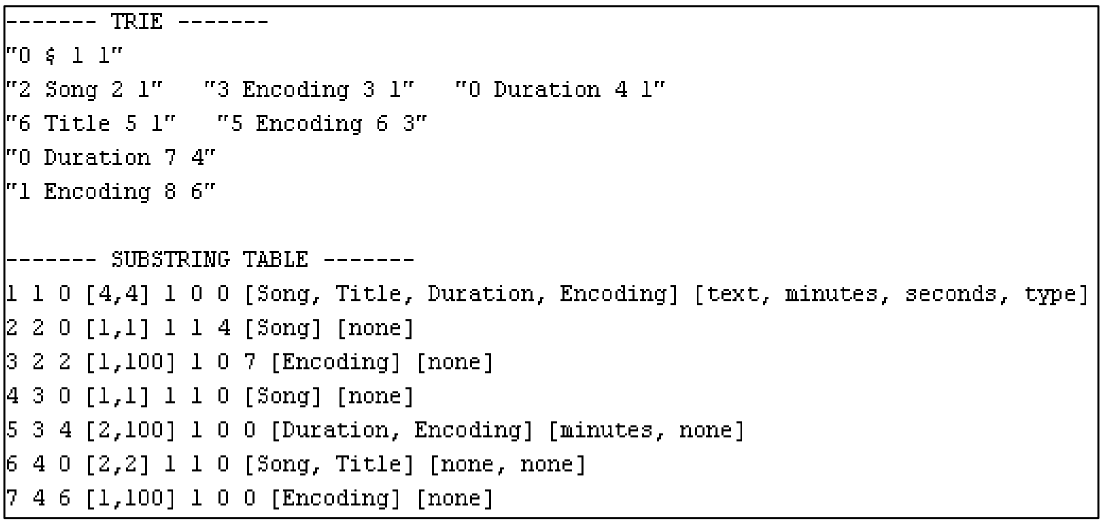
```
/Song[Title[text() = MoonChild]/Duration[@minutes<20][@seconds>10]/Encoding[@type]]
/Song//Encoding[@type = "Mpeg1-Layer 3"]
/Song//Duration[@minutes = 20]/Encoding
/Song/Title//Encoding
```

Figure 6.6 : The input XPE set

On the other hand, the substring table is a linked list of substring rows. The substring row shown in Figure 6.5b consists of 10 fields:

- *SubString*: is a Vector containing the substring of the simple XPE decomposition.
- *SID*: is the substring identifier.
- *xpeID*: is an integer with the identifier of the XPE.
- *ParentRow*: is the row number of the substring's father node in the substring tree.
- *RelLevel*: is an array with the relative level of the substring.
- *Rank*: is an integer with the position of the substring in its father Vector child.
- *NumChild*: is the number of the substring children in the substring tree.
- *Next*: is an integer with the row number corresponding to the same substring.
- *Attribute*: is a linked list which contains the attribute filters of the substring.
- *Text*: is a linked list which contains the text filters of the substring.

As an example to see how the XTrie builder works, consider the four XPath expressions, shown in Figure 6.6, as an input for the XTrie system. The XTrie builder constructs the Trie and the Substring Table, as depicted in Figure 6.7. Every node in Trie consists of the substring pointer [9], the element name, the node number and the max suffix pointer [9] respectively. Every row in the substring table contains the row number of the substring, the XPE identifier, the row number of the parent substring, the relative level of the substring, the rank, the number of substring's children, the



```
------- TRIE -------
"0 $ 1 1"
"2 Song 2 1"   "3 Encoding 3 1"   "0 Duration 4 1"
"6 Title 5 1"   "5 Encoding 6 3"
"0 Duration 7 4"
"1 Encoding 8 6"

------- SUBSTRING TABLE -------
1 1 0 [4,4] 1 0 0 [Song, Title, Duration, Encoding] [text, minutes, seconds, type]
2 2 0 [1,1] 1 1 4 [Song] [none]
3 2 2 [1,100] 1 0 7 [Encoding] [none]
4 3 0 [1,1] 1 1 0 [Song] [none]
5 3 4 [2,100] 1 0 0 [Duration, Encoding] [minutes, none]
6 4 0 [2,2] 1 1 0 [Song, Title] [none, none]
7 4 6 [1,100] 1 0 0 [Encoding] [none]
```

Figure 6.7 : The Trie and the Substring Table produced by XTrie builder

next pointer, the substring name and the attribute/text filters of the substring respectively.

6.2.5 XML Document Parser

The XML document parser takes an XML document and sends parsing events to the XTrie index. We have used SAX v2 [33] interface shipped in Sun Java package [23]. The followings are SAX 2 handler interfaces:

- *public void startDocument ().*This notifies the start of XML document. The SAX parser will invoke this method at the beginning of the document; there will be a corresponding *endDocument* event when the parser has finished the parsing of the document.
- *public void startElement (String uri, String localName, String qName, Attributes attributes).*This notifies the start of an element. The SAX parser will invoke this method at the beginning of every element in the XML document; there will be a corresponding *endElement* event for every *startElement* event (even when the element is empty). All of the element's contents will be reported in order, before the corresponding *endElement* event. It uses *uri* as the namespace of the XML document, *localName* as the local name (without

prefix), *qName* as the qualified name (with prefix), and *attributes* as the specified or defaulted attributes.

- *public void characters (char[] ch, int start, int length).* The parser will call this method to report each chunk of character data. The parser may return all contiguous character data in a single chunk, or it may split the returning data into several chunks; however, all of the characters in any single event must come from the same external entity so that the locator provides useful information. It uses XML documents character data from *ch* array, using *start* as the start position in the array and *length* as the number of characters to read from the array.

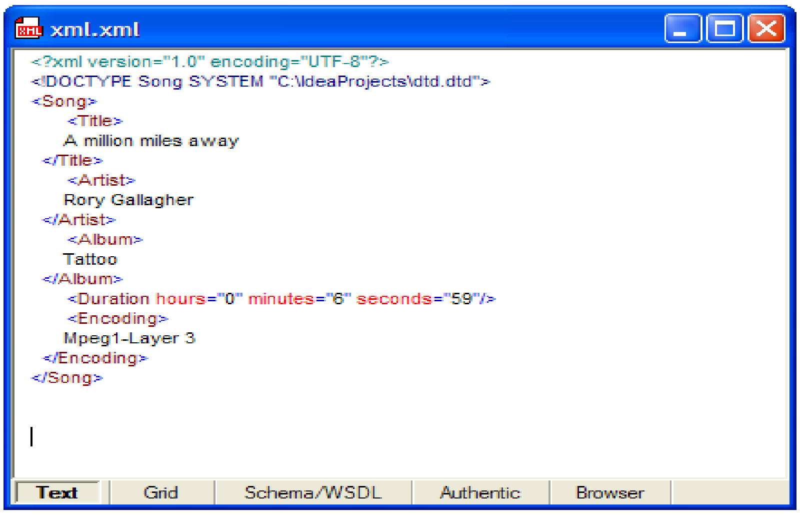
6.2.6 XTrie Matching Algorithms

We have implemented two algorithms for the XPE matching over streaming XML documents. These algorithms are:

- *Eager* XTrie algorithm [9].
- *Lazy* XTrie algorithm [9].

Figure 6.8a shows the attributes which are used by the Eager XTrie algorithm. These attributes are:

- *STsize*: is the total number of rows in the substring table.
- *Lmax*: is the maximum depth of the XML document.
- *NumOfXPEs*: is the total number of XPEs being indexed.
- *B*: is an integer array used by the matching algorithm to detect redundant matchings.
- *C*: is a bit array used by the matching algorithm to detect redundant matchings.
- *R*: is a Vector which contains the identifiers of the matched XPEs.
- *Node*: is a Vector which contains the Trie nodes that match an XML element.
- *N*: is the current Trie node.
- *l*: is the current document level.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Song SYSTEM "C:\IdeaProjects\dtd.dtd">
<Song>
     <Title>
    A million miles away
  </Title>
     <Artist>
    Rory Gallagher
  </Artist>
     <Album>
    Tattoo
  </Album>
     <Duration hours="0" minutes="6" seconds="59"/>
     <Encoding>
    Mpeg1-Layer 3
  </Encoding>
</Song>
```

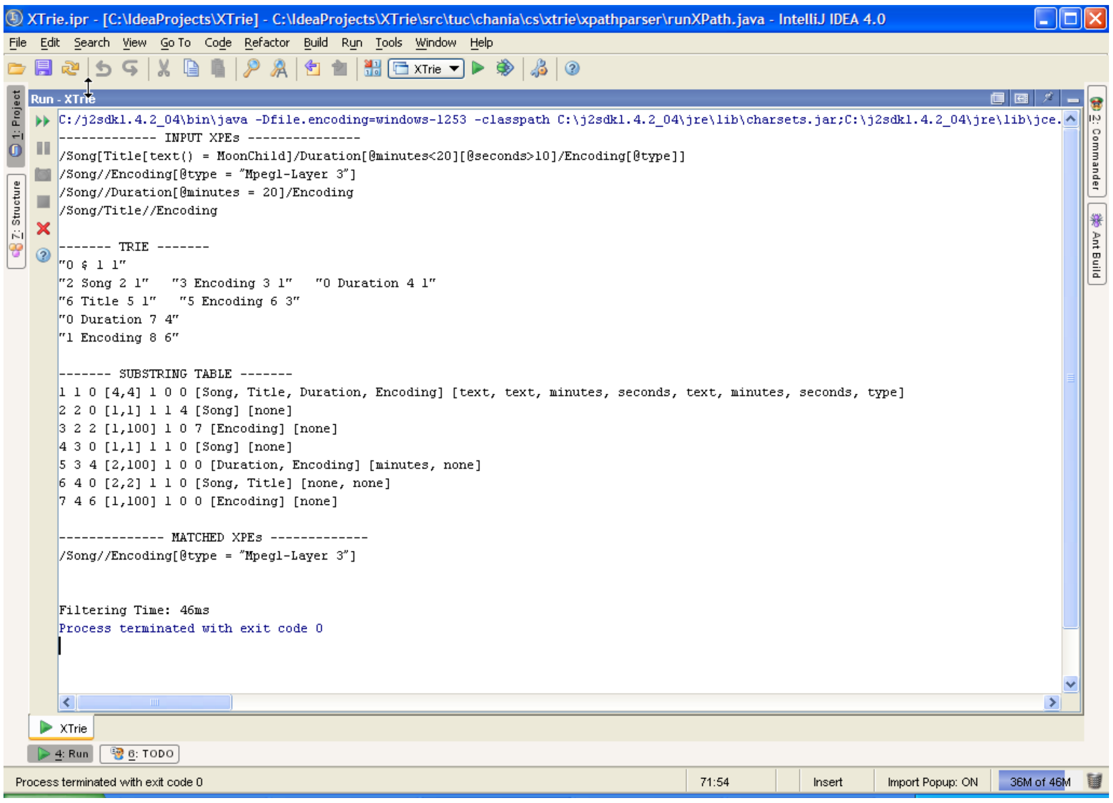
Figure 6.8 : The XML Document as input of the XTrie system

- *XMLattributes*: is a Vector which contains the attributes of an XML element given by the SAX parser.
- *XMLtext*: is a Vector which contains the enclosed text of an XML element given by the SAX parser.
- *id*: is the current XPE identifier.

The attributes used by the Lazy XTrie algorithms, as shown in Figure 6.7b, are similar to those used by the Eager XTrie. Moreover, Lazy XTrie uses two more attributes:

- *Ssize*: is the total number of rows in the substring table containing only distinct substrings.
- *M*: is a bit array used for the bookkeeping of the matched substrings.

The matching algorithm takes as an input the XTrie structure (Eager or Lazy) and the SAX events (element, attributes, text) produced by the XML document parser. Then it uses the attributes which are described before and returns the identifiers of the matched XPEs over the XML document.


XTrie.ipr - [C:\IdeaProjects\XTrie] - C:\IdeaProjects\XTrie\src\tuc\chania\cs\xtrie\xpathparser\runXPath.java - IntelliJ IDEA 4.0
File Edit Search View Go To Code Refactor Build Run Tools Window Help
XTrie
Run - XTrie
1: Project
7: Structure
2: Commander
Ant Build
XTrie
4: Run
6: TODO
71:54
Insert
Import Popup: ON
36M of 46M

```
C:/j2sdk1.4.2_04\bin\java -Dfile.encoding=windows-1253 -classpath C:\j2sdk1.4.2_04\jre\lib\charsets.jar;C:\j2sdk1.4.2_04\jre\lib\jce.
------------- INPUT XPEs ---------------
/Song[Title[text() = MoonChild]/Duration[@minutes<20][@seconds>10]/Encoding[@type]]
/Song//Encoding[@type = "Mpeg1-Layer 3"]
/Song//Duration[@minutes = 20]/Encoding
/Song/Title//Encoding

------- TRIE -------
"0 $ 1 1"
"2 Song 2 1"   "3 Encoding 3 1"   "0 Duration 4 1"
"6 Title 5 1"   "5 Encoding 6 3"
"0 Duration 7 4"
"1 Encoding 8 6"

------- SUBSTRING TABLE -------
1 1 0 [4,4] 1 0 0 [Song, Title, Duration, Encoding] [text, text, minutes, seconds, text, minutes, seconds, type]
2 2 0 [1,1] 1 1 4 [Song] [none]
3 2 2 [1,100] 1 0 7 [Encoding] [none]
4 3 0 [1,1] 1 1 0 [Song] [none]
5 3 4 [2,100] 1 0 0 [Duration, Encoding] [minutes, none]
6 4 0 [2,2] 1 1 0 [Song, Title] [none, none]
7 4 6 [1,100] 1 0 0 [Encoding] [none]

-------------- MATCHED XPEs -------------
/Song//Encoding[@type = "Mpeg1-Layer 3"]


Filtering Time: 46ms
Process terminated with exit code 0
```

```
Process terminated with exit code 0
```

Figure 6.9 : The output of the XTrie system

Now we can see a final example on how the XTrie system works. The XTrie system takes as an input the set of XPEs in Figure 6.6 and an XML document, shown in Figure 6.8, and produces the output as shown in Figure 6.9. (The only matched XPE in this example is the */Song//Encoding[@type = "Mpeg1-Layer 3"]).*

6.3 Summary

In this chapter, we presented the parts of the XTrie architecture and we described in detail the data structures and their functionality of our implementation. Then, we gave some examples to show how the XTrie system works. In the final chapter, we give our conclusions and discuss our suggestions to further work on the XTrie system.

Chapter 7

7 Conclusions and Future work

In this chapter, we present our conclusions and summarize our work in this dissertation. Also, we discuss how our work can be extended to support more features and how the XTrie indexing scheme can be used by data-share systems.

7.1 Concluding Remarks and Summarizing

In this work, we dealt with the problem of filtering streaming XML documents with XPath expressions. The main goal was to implement the XTrie structures and algorithms for efficient XML filtering. The XTrie [9] system, provides efficiency and scalability, has low space requirements and offers high throughput. Those features make it especially attractive for large scale distributed systems over the internet such as publish/subscribe systems.

Initially, we made an introduction in the area of publish/subscribe systems and we presented the XPE retrieval problem [9].

Moreover, we described the existing types of data models and their query languages and focused on XML data model and XPath query language. We explained why SAX [33] parser is better than DOM [41] for XML data streams filtering. Furthermore, we presented the XML DTD [42] and how can be used for the validity of the XML documents. Also, we presented the basic features of the XML syntax and we described the basic fragment of XPath used by XTrie.

Then, we discussed some alternative systems for XML processing. We made a distinction between the systems for XML filtering from those for XML (streaming and non-streaming) querying. We discussed some of their advantages and disadvantages according to XTrie. Moreover, we referred to several other systems related with the XML streaming problem.

We presented some important definitions about the basic features used by XTrie. We explained how an XPE can be represented as a rooted tree and match an XML document [9]. Further, we described the mechanisms for decomposing XPEs into sequences of XML element names (i.e., *substrings*), and explained how these substrings can be organized into substring-trees [9]. Then, we continued by discussing some important concepts of matching based on substring-trees [9].

We described in detail the XTrie index structure and matching algorithm [9] for the ordered matching model. We explained how the Trie [9] and the Substring table [9] can be used for effective matching over a streaming XML document. Moreover, we explained step by step all the algorithms [9] in the XTrie indexing scheme. Also, we discussed an optimized variant [9] of XTrie and explained how the matchings algorithms work. We continued by presenting two maintenances [9], one for XPE insertions into the XTrie structure and one for XPE deletions from the XTrie structure.

Finally, we presented our implementation in detail giving the structures and the algorithms written in Java [23]. We continued by presenting the basic steps of the XTrie model:

- Parse the XPath expression.
- Organize the XPath nodes in a tree and find the relative level of every node.
- Find the simple decomposition of the XPEs with a preorder traversal of the XPE tree.
- Decompose XPath tree in substring tree and find the relative level, the rank and the number of children of every node.
- Construct the Trie and the Substring table.
- Parse the XML document with the SAX [33] parsing interface.

- Run the matching algorithm (lazy or eager) using the events produced by the SAX parser and return the identifiers for the matched XPEs over the XML document.

7.2 Future Work

In this dissertation we implemented the XTrie model which solves the problem of filtering XML documents with XPath expressions. The fragment of XPath expressions used by XTrie is limited, so it can be extended to support more features of the XPath recommendations [46], such us aggregation functions [46] (sum, count, avg etc.), reverse axes [46] (preceding-sibling and forward-sibling), position functions [46] (pos() and last()), composite XPEs [9] (Boolean combinations of two or more XPEs) and absolute path expressions in predicates [9] (this implementation of XTrie supports only relative path expressions in predicates). Moreover our implementation of the XTrie indexing scheme supports only the ordered matching model. Thus, it can be extended to handle unordered matchings [9] and hybrid matchings [9] (combination of ordered and unordered matchings).

On the other hand, we can extend our work by implementing a simple service-client system, running in a network, which can use the XTrie model. This system, on top of the service, could run a file sharing application giving to the users the ability of:

- Publish their files (as XML documents) so that other users may see and download them.
- Query the system (with XPath expressions) to search for files on the whole network.
- Subscribe with a profile (written in XPath).

In addition, we could use the XTrie model in more complex file sharing systems. For example, we can use XTrie over distributed peer-to-peer systems such as P2P-DIET [20] (which has been implemented in the Technical University of Crete), SIENA [8] and more.

Bibliography

[1] Aguilera, M.K., Strom, R.E., Sturman, M., Astley, D.C., Chandra, T. D. Matching events in a content-based subscription system. *In Proceedings ACM (PODC)*, Atlanta, Ga., USA, pages 53–61, 1999.

[2] Altinel, M., Franklin, M.J. Efficient filtering of XML documents for selective dissemination of information. *In Proceedings VLDB*, pages 53–64, 2000.

[3] Avila-Campillo, I., Raven, D., Green, T., Gupta, A., Kadiyska, Y., Onizuka, M., and Suciu, D. An XML Toolkit for Light-weight XML Stream Processing. http://www.cs. washington.edu/homes/suciu/XMLTK/, 2002.

[4] Avnur, R. and Hellerstein, J. M. Eddies: Continuously Adaptive Query Processing. *In the 19th ACM SIGMOD International Conference on Management of Data*, pages 261-272, 2000.

[5] Barton, C. M., Charles, P. G., Goyal, D., Raghavachari, M., Josifovski, V., and Fontoura, M. F. Streaming XPath Processing with Forward and Backward Axes. *In the 18th International Conference on Data Engineering*, 2003.

[6] Becker, O., Cimprich, P., and Nentwich, C. Streaming Transformations for XML. http://www.gingerall.cz/stx, 2002.

[7] Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. Monitoring Streams: A New

Class of Data Management Applications. *In Proceedings of the 28th International Conference on Very Large Data Bases*, pages 215-226, 2002.

[8] Carzaniga, A., Rosenblum, D.S, Wolf, A.L. Design and evaluation of a wide-area event notification service. *ACM Trans. Computer Systems*, 19(3):332–383, 2001.

[9] Chan, C. Y., Felber, P., Garofalakis, M. N., and Rastogi, R. Efficient Filtering of XML Documents with XPath Expressions. *In the 18th International Conference of Data Engineering*, pages 235-244, 2002.

[10] Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. Alternation. *Journal of the ACM (JACM),* 28(1):114-133, 1981.

[11] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *In the 19th ACM SIGMOD international conference on Management of data*, pages 379-390, 2000

[12] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. Scalable Distributed Stream Processing. *In the First Biennial Conference on Innovative Database Systems*, 2003.

[13] Diao, Y., Fischer, P., and Franklin, M. J. YFilter: Efficient and Scalable Filtering of XML Documents. *In the 18th International Conference of Data Engineering*, pages 341-344, 2002.

[14] Diaz A.L., Lovell D. XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, 1999.

[15] Feng Peng and Sudarshan S. Chawathe. XSQ: A Streaming XPath Engine. Technical Report CS-TR-4493 (UMIACS-TR-2003-62). Computer Science Department, University of Maryland, College Park, Maryland 20742, 2003.

[16] Fernandez, M. and Simeon, J. Galax. http://db.bell-labs.com/galax/, 2002.

[17] Gottlob, G., Koch, C., and Pichler, R. Efficient algorithms for processing XPath queries. *In Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, 2002.

[18] Green, T. J., Miklau, G., Onizuka, M., and Suciu, D. Processing XML streams with Deterministic Automata. *In the 9th International Conference on Database Theory*, Siena, Italy, pages 173-189, 2003.

[19] Hoffmann, C. M. and O'Donnell, M. J. Pattern matching in trees. *Journal of the ACM (JACM)*, 29(1): 68-95, 1982.

[20] Idreos, S. P2P-DIET: A query and notification service based on mobile agents for rapid implementation of P2P applications. Technical report TUC-ISL-01-2003, Intelligent Systems Laboratory, Dept. of Electronic and Computer Engineering, Technical university of Crete, June 2003.

[21] Intel netStructure XML accelerators. http://www.intel.com/netstructure/products/xml_accelerators.htm, 2000.

[22] Java 2 SDK Home Page. http://java.sun.com/j2se.

[23] Java Sun Home Page. http://java.sun.com.

[24] Katz, H. XQEngine. http://www.fatdog.com, 2002.

[25] Kay, M. H. SAXON: an XSLT processor. http://saxon.sourceforge.net/, 2002.

[26] Kilpel, P. Tree matching problems with applications to structured text databases. Ph.D. thesis, Dept. of Computer Science, University of Helsink, 1992.

[27] Knuth D.E. The art of computer programming: sorting and searching, vol. 3, 2nd edn. Addison Wesley, Reading, Mass., USA, 1998.

[28] Lakshmanan, L. V. and Sailaja, P. On Efficient Matching of Streaming XML Documents and Queries. *In the 8th International Conference on Extending Database Technology*, Prague, Czech Republic, pages 142-160, 2002.

[29] Liu, L., Pu, C., and Tang, W. Continual Queries for Internet Scale Event-Driven Information Delivery. *Knowledge and Data Engineering*, 11(4): 610-628, 1999.

[30] Liu, L., Pu, C., and Tang, W. Webcq-detecting and delivering information changes on the web. *In the 9th International Conference on Information and Knowledge Management*, pages 512-519, 2000.

[31] Ludascher, B., Mukhopadhayn, P., and Papakonstantinou, Y. A Transducer-Based XML Query Processor. *In the 28th International Conference on Very Large Data Bases*, Hong Kong, China, pages 227-238, 2002.

[32] Madden, S. and Franklin, M. J. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. *In the 18th International Conference of Data Engineering*, 2002.

[33] Megginson, D. SAX: a simple API for XML. http://www.megginson.com/SAX/, 2002.

[34] Miklau, G. and Suciu, D. Containment and Equivalence for an XPath Fragment. *In the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Madison, Wisconsin, pages 65-76, 2002.

[35] Olteanu, D., Kiesling, T., and Bry, F. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. Tech. Rep. PMS-FB-2002-12, Institute for Computer Science, Ludwig-Maximilians University, Munich, May 2002.

[36] Plale, B. and Schwan, K. dQUOB: Managing Large Data Flows by Dynamic Embedded Queries. *In the 9th IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, Pennsylvania, pages 263-270, 2000.

[37] Segall, B., Arnold, D., Boot, J., Henderson, M., Phelps, T. Content-based routing with Elvin4. *In AUUG2K*, Canberra, Australia, 2000.

[38] Segoufin, L. and Vianu, V. Validating Streaming XML documents. *In the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Madison, Wisconsin, pages 53-64, 2002.

[39] SQL Home Page. http://www.microsoft.com/sql.

[40] Tucker, P. A., Maier, D., and Sheard, T. Applying punctuation schemes to queries over continuous data streams. *Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society*, 26(1):33-40, March 2003.

[41] W3C Document object model (DOM) level 1 specification 1.0, 2nd edn., http://www.w3.org/TR/REC-DOM-Level-1/, 2002.

[42] W3C Extensible markup language (XML) 1.0, 2nd edn., http://www.w3.org/TR/REC-xml/, 2000.

[43] W3C Home Page. http://www.w3.org.

[44] W3C Hyper Text Markup Language (HTML) 4.0, http://www.w3.org/TR/WD-html40/, December 1997.

[45] W3C SGML. http://www.w3.org/Markup/SGML/, 1999.

[46] W3C XML path language (XPath) 1.0, http://www.w3.org/TR/xpath/, 1999.

[47] W3C XSL Working Group. XSL Transformations (XSLT) 2.0. W3C Working Draft, W3C, http://www.w3.org/TR/xslt20/, April 2002.

[48] Widom, J. The starburst active database rule system. *IEEE Transactions of Knowledge and Data Engineering*, 8(4):583-595, August 1996.

[49] Zdonik, S., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M., and q, H. B. The aurora and medusa projects. *Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society*, 26(1): 3-10, March 2003.