DOCUMENT PROVISOIRE

Thèse

Spécialité Mathématiques et Informatique

soutenue par

Grégory COBÉNA

Change Management of semi-structured data on the Web

Directeur de thèse : **Serge ABITEBOUL**

Table des matières

Table des figures

Chapter 1

Introduction

The new era of XML Web services is the most exciting ever for this industry and for customers worldwide.

Bill Gates

Le Web est une grande poubelle à ciel ouvert. Il faut savoir détecter l'importance de ce que l'on retient.

Serge Abiteboul

For me the fundamental Web is the Web of people. It's not the Web of machines talking to each other; it's not the network of machines talking to each other. It's not the Web of documents. Remember when machines talk to each other over some protocol, two machines are talking on behalf of two people.

Tim Berners-Lee

Context In the recent years, two events related to the Wold Wide Web have changed dramatically the way people can use it. One is the explosion of content, i.e. the increase of published documents and data on the web. From administrative forms to scientific or medical data, as well as home, travel and entertainment, the web has become the largest repository of content that is freely available since the beginning of mankind. The second event is the creation and acceptance of XML-related technologies. The use of XML and semi-structured data will enable the development of high quality services on the web.

Users are interested in gathering knowledge and data from the web. One may remark for instance that users often search for news. Newspaper sites (e.g. *LeMonde*, *CNN* or *TF1*) have had a remarkable success. Consider for instance a person who is interested in Art, or History. Even if there is already a very large amount of available knowledge on the topic, this person often wishes to subscribe to news magazines, mailing lists or newsletters to be regularly informed. *We believe that users are often interested as much (if not more) in changes of data, e.g. new data, than on the data itself.*

In this thesis, we present work on the topic of change-control. In particular our work considers change-control on the web, and change-control on semi-structured data. In other words, we consider data and their changes from a microscopic scale to a macroscopic scale. More precisely we consider data and changes at the scale of document elements (e.g. XML fragments), and at the scale of the Internet, the World Wide Web.

This translates into 4 aspects that we address in this thesis. The four aspects are as follows:

(i) Find data and sources of data. This is related for instance to our work on archiving the French web. The two main issues are the definition of the “frontier” of the French web, and the selection of valuable content to archive. As we will see further in this thesis, the issue of selection to obtain a reasonable level of “quality” of information is a key towards the success of any automatic processing of web data.

(ii) Monitor these documents and data through time. This is related to previous work on acquisition and refreshment of web data [79, 59]. Our work consists in understanding *when* documents changes, and *what* are the changes that occur. To improve the quality of our results, we choose to use as much as possible the structure of data: the structure of XML documents, and the structure of web sites where documents are found. In the context of documents from the web, a critical requirement is performance since it is necessary to scale to the size of the web.

(iii) Extract knowledge on the changing contents of documents, or on their changing metadata. This denotes the need to add semantic value to data and to the changes of data. This problem is illustrated in the first part of this thesis where our study on change detection analyzes various notions of "quality" of results. These are for instance minimality of the delta results, the support for *move* operations that enable a better identification of nodes through time than insert and delete, and the support of constraints from the DTDs in the spirit of database *keys*.

(iv) Extract knowledge on the changing relations between documents, or on the changing collections of documents. This is a typical field of information retrieval, and previous work is abundant on the topic of analyzing collections of documents [54, 62]. We propose a possible approach to this problem in the second part of the thesis by presenting an algorithm that computes the "importance" of pages on the web, and that adapts to the changes of the web.

Macroscopic changes. Let us first consider the macroscopic changes. The web is a large source of documents, and we believe that the change of its content is very valuable for several reasons. First, because new documents are added that continuously enrich the available knowledge. Second, because updates of pieces of information are made precisely to correct and improve the previous data. Third, because these changes that are made are information them-selves. For instance, each time a stock value changes, this is information.

Moreover, there are other reasons why it is important to learn about changes on the web.

One for instance is performance. The web contains today more than 4 billion pages, and, for instance, to monitor the web we must focus processing resources in the most efficient manner. This implies for instance that no redundant processing should be performed, and in particular we should focus on processing changing data and not all the data.

Another reason to learn about changes is that web pages tend to change and disappear very quickly. It is very important to have the possibility to find some information *as it was* at some previous time.

In a few words, we should say that the focus on changes on the web is necessary to improve (i) the performance of applications, (ii) the quality of information. Version management on the web is also necessary because it ensures that history can not be rewritten.

Microscopic changes. Let us now consider the microscopic changes, i.e. changes at the document level. The main aspect of change-control in documents is studying differences between several versions of a document. For instance, addresses that are modified in some address book.

Our work is focused on XML documents. Extensible Mark-up Language (XML) is a simple, very flexible text format derived from SGML (ISO8879). In the spirit of SGML and HTML, tags (e.g. `<name>Greg</name>`) are used to give a tree structure to the document. Today, XML [102] is a *de facto* standard for exchanging data both in academia and in industry. We also believe that XML is becoming a standard to model stored data.

With XML, documents are represented using a precise schema. However, as opposed to usual (e.g. relational) databases, the schema does not need to be strictly defined as a prerequisite to handle the data. The advantage of XML compared to flat text files is that it adds information that gives a structure to data. For instance, when a journalist writes an article to summarize a soccer game, the labeling structure of XML states for each sentence, and for each word, whether it is the name of a player, or the name of the referee, or the town where the game took place. Then, it is possible to write programs that retrieve this information and process user queries, for instance finding each soccer player that scored a goal in *Stade de France*.

XML and semi-structured data form an essential component to enable a more efficient distribution of knowledge and services on the web. The success of XML, which seems certain at this point, marks the beginning of a new era for knowledge and services on the Internet.

We present our work to detect, store and monitor changes on XML documents. To do so, we present an algorithm to detect changes in XML documents, and we present a model for representing them in XML. We also present a comparative study on this topic.

An important aspect is to consider the quality of results and the semantics of data. Consider for instance a document representing the list of employees and their phone numbers in some company. The differences between two versions of the document may be interpreted in different ways. For instance, the department of human resources might track the changing phone numbers for each employee. On the other hand, the maintenance department is interested in the list of phones, and their technical application considers that for some phone, it is the name of the employee that changes. The two interpretations lead to different representations of changes, although the actual changes in the document may be the same. It is then necessary to integrate the semantics of data and their changes in the algorithms and models.

In this thesis, we will consider both the performance and quality for change-control of semi-structured data.

Note that change-control of XML documents comes down to comparing two versions of some XML document. This same technology may be used to find differences (and similarities) between two different XML documents. For instance, what are the differences between the vendor's XML description of two car models.

Organization. The thesis is organized in 3 parts as follows:

In the first part, we present our work in change-control at the microscopic scale, that is inside XML documents. First, we present an algorithm that, given two versions of a document, detects changes that occurred between them, and constructs a delta that transforms one version of the document into the other. Then, we propose a formal model for representing changes of XML documents in XML. In the third chapter, we present a state of the art in the field of change detection and representation, namely a comparative study that we conducted recently. Finally, we will present our work in the context of the Xyleme project. This work consists in integrating the work on XML document changes into a web crawler to be able to monitor document changes on the web.

In the second part, we present our work on change-control at the macroscopic scale, that is at the scale of the web. Indeed, we will first study the graph of the web, and show an algorithm that can be used to compute the importance of pages online, i.e. while the web is crawled. It adapts dynamically to the changes of the graph of the web. Then, we illustrate this topic by describing some work in the context of “web archiving”. The web is a more and more valuable source of information and this leads national libraries (e.g. the French national library) and other organizations to archiving (portions of) the web. This is in the spirit of their work on other medias, for instance archiving books and newspapers.

The last part is a conclusion.

Chapter 2

Preliminaries

To further understand the context of this work, we first present some fundamental notions that will be used through this thesis. In particular, we give an overview of XML and DTDs and mention briefly XMLSchema, SOAP and Web Services. More insights may be found in [3, 104, 105].

XML

XML is a format to represent semi-structured data. Semi-structured data may be defined by comparing "structured" data and "un-structured" data.

- On one hand, un-structured data is typically represented by some text. For instance, consider a letter from your bank containing important account informations. Understanding the text itself is necessary to find the relevant information. For instance, if the letter is written in Japanese, I would not be able to find any information. It is a difficult task to create programs that understand and use such data.

- On the other hand, structured data is for instance the data of a relational database. It is typically in tables, with rows and columns, where each cell contains a precise piece of information. In a bank account statement there is typically a table with rows for each banking operation, and columns for the transaction date, the amount of money, the operation descriptor, and the account balance. It is easy to write programs that use the data, for instance a program that computes how much money is spent each week on average. The problem is that it is not always easy to put data in a structured format, for instance it would be difficult to put a news article in a relational database.

Semi-structured data lay somewhere in between. In this thesis, we focus on a particular kind of semi-structured data model, namely XML.

An XML document is represented as a text file with opening and closing tags that give a tree structure to the data. XML does not use predefined tags. It has the advantage of text files that it is easy to represent a document as an XML document. It has the advantage of structured data that there is a structure that can be used to validate a document (see DTD and XMLSchema [104]), to query the content and to give a specific semantic to each piece of data [103]. A typical example is as follows:

```
<letter>
   <from> A.B.C. Bank
     <address>
       100 El Camino Real, 94025 Menlo Park
     </address>
   </from>
   <to>
     Jacques Dupond
     <address>
       21 Edgewater Bvd, 94404 Foster City
     </address>
   </to>
   <date>
     10/10/2003
   </date>
   <body>
     Dear Sir,

     Due to ...
     And ...
     We inform you that your account has been charged with <amount op-
eration="mensual-fee">$8</amount>
     ...
   </body>
</letter>
```

An essential aspect of XML is the tree structure of XML data. Indeed, while the textual content is represented at the leaves of the tree, the internal nodes, namely element nodes, represent the structure of the data. They facilitate the usage of data and the understanding of its precise semantics. Applications and users can more easily understand the meaning of each piece of the document. Moreover, when the document changes, applications and users can more easily understand the meaning of these changes.

XML documents may be easily queried. For instance: find the name and address of all customers to which we sent two letters for charging the mensual fee the same month. This is not easy with textual data.

Note that this kind of query was already possible with relational databases. However, the main problem is that it is difficult to organize the contents and knowledge of a company in a relational database, because each piece of information is slightly different from the others, and structured databases require very precise formats and schema for their data.

XML has been adopted as a standard format for exchanging data. The coming of Web Services, WSDL and SOAP based on XML, confirms the acceptance of XML. We believe that XML is also becoming a standard for storing data, although this is somewhat more controversial.

XML vs. HTML. The HTML format is the most common format for displaying web pages. HTML stands for Hyper Text Markup Language [104, 105]. An HTML file is a text file containing small markup tags. The markup tags tell the web browser how to display the page.

The extreme simplicity of HTML and its flexibility has played an important role in the success and the rapid development of the Internet by enabling quick design of web pages. In particular, web browsers have been very tolerant towards syntactical and structural errors in the HTML source of the web pages they display. The undesired consequence is that most HTML documents have no real structure and can not be used by data-intensive applications. Today, as we will see later in this thesis, the web is much like HTML: it is a large container of knowledge, with little or no structure and no explicit way to find valuable information among a wide volume of "junk" [6].

While HTML enables a graphical representation of web pages that permits humans to easily read documents, it does not facilitate the use and exchange of data by programs.

XSL. Although this is not necessary for the comprehension of this thesis, we mention briefly the role played by XSL towards the XML web. The Extensible Stylesheet Language, XSL, is a language for expressing style sheets [104, 105]. Because XML does not use predefined tags (we can use any tags we want), the meanings of these tags are not understood: `<table>` could mean an HTML table, a piece of furniture, or something else. Therefor, a generic browser does not know *a priori* how to display an XML document. Some extra information is needed to describe how the document should be displayed; and that is XSL.

For instance, XSL may be used to define the font style and size for the title of a document, a specific font and layout for a part of some document, and so on. We have mentioned previously that most web pages on the web are HTML. However, the content is in fact more and more stored as XML or structured data, and is often exported as XML from a database. Then, XSL stylesheets are used on the server, and using their layout and presentation information, the content is published as HTML pages on the web.

Other languages can be used to display HTML pages based on program data or database queries. The most popular are *PhP*, *Sun JSP*, *Microsoft ASP*.

XSLT XSL Transformations, XSLT, is a language for transforming XML documents into other XML documents. XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary. XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL.

DTD and XMLSchema. The purpose of a Document Type Definition (see DTD [104]) is to define the legal building blocks of an XML document [105]. It defines the document structure with a list of legal elements, more precisely it specifies the legal tags and the structure of the elements of these tags. With DTD, each XML file may carry a description of its own format with it. In particular, groups of people may agree to use a common DTD for exchanging data. Applications can use a standard DTD to verify that the data they receive from the outside world is valid. One may also use a DTD to validate his own data.

An important aspect of DTDs that we use in Chapter 3 is the possibility to define ID attributes. Attributes are pieces of information (a name and a value) that are attached to *element* nodes of XML documents. Specifying in the DTD that an attribute is an `ID Attribute` indicates that the value of this specific attribute is a unique identifier for the element node. This is in the spirit of *keys* in databases.

The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD. It is likely that very soon XML Schemas will be used in most web applications as a replacement for DTDs. Here are some reasons:

- XML Schemas propose a richer typing than DTDs,
- XML Schemas are written in XML,
- XML Schemas support data types,
- XML Schemas support namespaces and other features that facilitate their use in complex settings.

XPath XPath is a syntax for accessing parts of an XML document. More precisely, XPath uses path expressions to identify nodes in an XML document. Some simple XPath expressions look very much like the expressions you see when you work with a computer file system:

```
company/listOfEmployees/employee/name
```

Web Services: SOAP, WSDL. Briefly, Web Services offer the possibility to execute remote function calls on the Internet. SOAP is the remote call specification, and WSDL in the method interface specification.

SOAP is a lightweight XML-based protocol for exchange of information in a decentralized, distributed environment. The advantage of SOAP compared to existing technologies (such as RPC, DCOM, CORBA) is that it uses HTTP and thus it is easily integrated in a large-scale web environment with firewalls and proxys. SOAP provides a way to communicate between applications running on different operating systems, with different technologies and programming languages.

WSDL stands for Web Services Description Language. WSDL is an XML format for describing network services. In the spirit of CORBA Interface Description Language [87], it specifies the operations (or methods) of the service. It also specifies the messaging protocol, mainly SOAP, as well as the web location (URL) of the service.

Part I

Controlling Changes in XML documents

Chapter 3

An XML diff algorithm: XyDiff

Abstract. *We present a* diff *algorithm for XML data. This work is motivated by the support for change control in the context of the Xyleme project that is investigating dynamic warehouses capable of storing massive volume of XML data. Because of the context, our algorithm has to be very efficient in terms of speed and memory space even at the cost of some loss of "quality". Also, it considers, besides insertions, deletions and updates (standard in diffs), a* move *operation on subtrees that is essential in the context of XML. Intuitively, our* diff *algorithm uses signatures to match (large) subtrees that were left unchanged between the old and new versions. Such exact matchings are then possibly propagated to ancestors and descendants to obtain more matchings. It also uses XML specific information such as ID attributes. We provide a performance analysis of the algorithm. We show that it runs in average in linear time vs. quadratic time for previous algorithms. We present experiments on synthetic data that confirm the analysis. Since this problem is NP-hard, the linear time is obtained by trading some quality. We present experiments (again on synthetic data) that show that the output of our algorithm is reasonably close to the "optimal" in terms of quality. Finally we present experiments on a small sample of XML pages found on the Web.*

In the context of the Xyleme project [117], some preliminary work on XML diff was performed by Amélie Marian. When she left for Columbia University, I took over the work on XML diff. This section presents my algorithm for XML diff. It was clearly influenced by original ideas by Abiteboul and Marian.

The algorithm has been presented in [35].

3.1 Introduction

Users are often not only interested in the current value of data but also in changes. Therefore, there has been a lot of work around *diff* algorithm for all kinds of data. With the Web and standards such as HTML and XML, tree data is becoming extremely popular which explains a renewed interest for computing changes in tree-structured data. A particularity of the Web is the huge volume of data that has to be processed. For instance, in the Xyleme project [117, 119], we were lead to compute the *diff* between the millions of documents loaded each day and previous versions of these documents (when available). This motivates the study of an extremely efficient, in terms of speed and memory space, *diff* algorithm for tree data.

As mentioned above, the precise context for the present work is the Xyleme project [119] that is studying and building a *dynamic World Wide XML warehouse*, i.e., a data warehouse capable of storing massive volume of XML data. XML, the new standard for semistructured data exchange over the Internet [102, 3], allows to support better quality services and in particular allows for *real* query languages [45, 91] and facilitates semantic data integration. In such a system, managing changes is essential for a number of reasons ranging from traditional support for versions and temporal queries, to more specific ones such as index maintenance or support for query subscriptions. These motivations are briefly considered in Section 3.2.

The most critical component of change control in Xyleme is the *diff* module that needs to be extremely efficient. This is because the system permanently receives XML data from the Web (or internal) crawlers. New versions of the documents have to be compared to old ones without slowing down the whole system.

Observe that the *diff* we describe here is for XML documents. It can also be used for HTML documents by XML-izing them, a relatively easy task that mostly consists in properly closing tags. However, the result of *diff* for a true XML document is semantically much more informative than for HTML. It includes semantic pieces of information such as the insertion of a new product in a catalog.

Intuitively, our algorithm works as follows. It tries to detect (large) subtrees that were left unchanged between the old and new versions. These are matched. Starting from there, the algorithm tries to match more nodes by considering ancestors and descendants of matched nodes and taking labels into consideration. Our algorithm also takes advantage of the specificities of XML data. For instance, it knows of attributes and attribute updates and treat them differently from element or text nodes. It also takes into account ID attributes to match elements. The matching of nodes between the old and new version is the first role of our algorithm. Compared to existing *diff* solutions such as [56, 78], our algorithm is faster and has significantly better matchings.

The other role of our algorithm is the construction of a representation of the changes using a *delta*. We use the *delta* representation of [69] that is based on inserts, deletes, updates and moves. For completeness, we present it in Section 3.4. Given a matching of nodes between the two documents, a *delta* describes a representation of changes from the first to the second. A difficulty occurs when children of a node are permuted. It is computationally costly to find the *minimum* set of move operations to order them.

We show first that our algorithm is “correct” in that it finds a set of changes that is sufficient to transform the old version into the new version of the XML document. In other words, it misses no changes. Our algorithm runs in $O(n * log(n))$ time vs. quadratic time for previous algorithms. Indeed, it is also noticeable that the running time of our algorithm significantly decreases when documents have few changes or when specific XML features like ID attributes are used. In Section 3.3, we recall that the general problem is NP-hard. Therefore, to obtain these performance we have to trade-in something, an ounce of “quality”. The delta’s we obtain are not “minimal”. In particular, we may miss the best match and some sets of move operations may not be optimal. It should be observed that any notion of minimality is somewhat artificial since it has to rely on some arbitrary choice of a distance measure. We present experiments that show that the *delta*’s we obtain are of very good quality.

There has been a lot of work on *diff* algorithms for strings, e.g., [65, 40, 106], for relational data, e.g., [64], or even for tree data, e.g., [108, 25]. The originality of our work comes from the particular nature of the data we handle, namely XML, and from strict performance requirements imposed by the context of Xyleme.

Like the rest of the system, the *diff* and the versioning system are implemented in C++, under Linux, with Corba for communications. Today, it is freely available on the Web in open-source [34]. It has been used by several groups over the world. In particular, it is included in the industrial product of Xyleme SA [119].

We performed tests to validate our choices. We briefly present some experimentation. The results show that the complexity of our algorithm is indeed that determined by the analysis, i.e., quasi linear time. We also evaluate experimentally the quality of the *diff*. For that, we ran it on synthetic data. As we shall see, the computed changes are very close in size to the synthetic (perfect) changes. We also ran it on a small set of real data (versions of XML documents obtained on the web). The size is comparable to that of the Unix Diff. This should be viewed as excellent since our description of changes typically contains much more information than a Unix Diff. We also used the *diff* to analyze changes in portions of the web of interest, e.g., web sites described as XML documents (Section 3.6).

We present motivations in Section 3.2 and consider specific requirements for our *diff*. A description of the change model of [69] is given in Section 3.4. We mention previous *diff* algorithms in Section 3.3. Note that an extensive state of the art is presented in Chapter 5, where we conduct a benchmark for XML change detection. In Section 3.5, we present the XyDiff algorithm, and its analysis. Measures are presented in Section 3.6. The last section is a conclusion.

3.2 Motivations and requirements

In this section, we consider motivations for the present work. Most of these motivations for changes detection and management are similar to those described in [69].

As mentioned in the introduction, the role of the *diff* algorithm is to provide support for the control of changes in a warehouse of massive volume of XML documents. Detecting changes in such an environment serves many purposes:

- **Versions and Querying the past:** One may want to version a particular document [69], (part of) a Web site, or the results of a continuous query. This is the most standard use of versions, namely recording history. Later, one might want to ask a query about the past, e.g., ask for the value of some element at some previous time, and to query changes, e.g., ask for the list of items recently introduced in a catalog. Since the *diff* output is stored as an XML document, namely a *delta*, such queries are regular queries over documents.

- **Learning about changes:** The *diff* constructs a possible description of the changes. It allows to update the old version V_i and also to explain the changes to the user. This is in the spirit, for instance, of the Information and Content Exchange, ICE [110, 57, 61]. Also, different users may modify the same XML document off-line, and later want to synchronize their respective versions. The diff algorithm could be used to detect and describe the modifications in order to detect conflicts and solve some of them [39].

- **Monitoring changes:** We implemented a subscription system [84] that allows to detect changes of interest in XML documents, e.g., that a new product has been added to a catalog. To do that, at the time we obtain a new version of some data, we *diff* it and verify if some of the changes that have been detected are relevant to subscriptions. Related work on subscription systems that use filtering tools for information dissemination have been presented in [120, 10].

- **Indexing:** In Xyleme, we maintain a full-text index over a large volume of XML documents. To support queries using the structure of data, we store structural information for every indexed word of the document [8]. We are considering the possibility to use the *diff* to maintain such indexes.

To offer these services, the *diff* plays a central role in the Xyleme system. Consider a portion of the architecture of the Xyleme system in Figure 3.1, seen in a change-control perspective. When a new version of a document $V(n)$ is received (or crawled from the web), it is installed in the repository. It is then sent to the *diff* module that also acquires the previous version $V(n-1)$ from the repository. The *diff* modules computes a delta, i.e., an XML document describing the changes. This delta is appended to the existing sequence of delta for this document. The old version is then possibly removed from the repository. The alerter is in charge of detecting, in the document $V(n)$ or in the *delta*, patterns that may interest some subscriptions [84]. Efficiency is here a key factor. In the system, one of the web crawlers loads millions of Web or internal pages per day. Among those, we expect many to be XML. The *diff* has to run at the speed of the indexer (not to slow down the system). It also has to use little memory so that it can share a PC with other modules such as the Alerter (to save on communications).

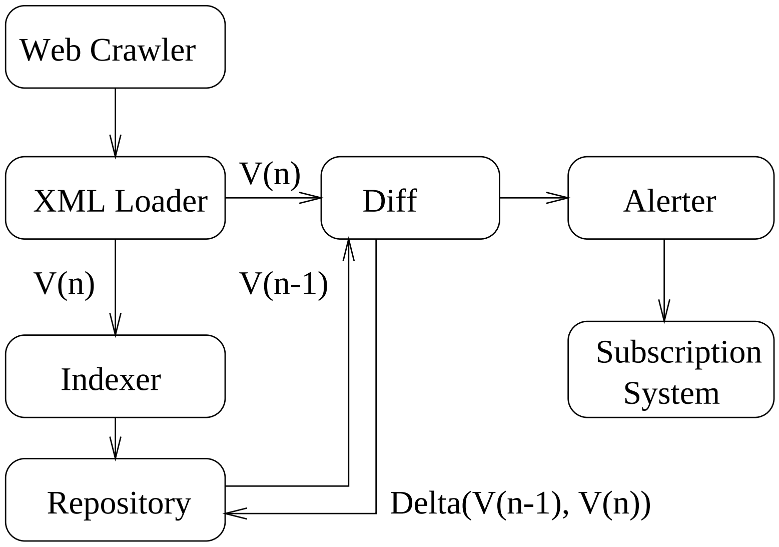

Figure 3.1: Xyleme-Change architecture

These performance requirements are essential. The context also imposes requirements for the deltas: they should allow (i) reconstructing an old version, and (ii) constructing the changes between some versions n and n'. These issues are addressed in [69]. The *diff* must be correct, in that it constructs a *delta* corresponding to these requirements, and it should also satisfy some quality requirements. Typically, quality is described by some minimality criteria. More precisely, the *diff* should construct a minimum set of changes to transform one version into the next one. Minimality is important because it captures to some extent the semantics that a human would give when presented with the two versions. It is important also in that more compact deltas provide savings in storage. However, in our context, it is acceptable to trade some (little) minimality for better performance.

We will see that using specificities of the context (in particular the fact that documents are in XML) allows the algorithm to obtain changes that are close to the minimum and to do that very efficiently. The specific aspects of XyDiff algorithm are as follows:

- Our *diff* is, like [24, 25], tailored to tree data. It also takes advantage of specificities of XML such as ID attributes defined in the DTD, or the existence of labels.
- The *diff* has insert/delete/update operations as in other tree *diff* such as [25], and it also supports a *move* operation as in [24]. The *move* allows to move an XML (possibly large) subtree.

3.3 State of the art

In a standard way, the *diff* tries to find a minimum *edit script* between the versions at time t_{i-1} and t_i. The basis of edit distances and minimum edit script is the string edit problem [11, 65, 40, 106]. Insertion and deletion correspond to inserting and deleting a symbol in the string, each operation being associated with a cost. Now the string edit problem corresponds to finding an edit script of minimum cost that transforms a string x into a string y. A most standard algorithm for the problem works as follows. The solution is obtained by considering prefixes substrings of x and y up to the i-th symbol, and constructing a directed acyclic graph (DAG) in which path $cost(x[1..i] \rightarrow y[1..j])$ is evaluated by the minimal cost of these three possibilities:

$$\begin{array}{l} cost(delete(x[i])) + cost(x[1..i-1] \rightarrow y[1..j]) \\ cost(insert(y[j])) + cost(x[1..i] \rightarrow y[1..j-1]) \\ cost(subst(x[i], y[j])) + cost(x[1..i-1] \rightarrow y[1..j-1]) \end{array}$$

Note that for example $subst(x[i], y[j])$ is zero when the symbols are equals. The space and time complexity are $O(|x| * |y|)$.

XML documents can represented as strings. Thus, string detection algorithm may be applied to XML documents based on their string representation. But this does not take into account the tree structure of the document. It is then possible to do some post-processing of the result in order to obtain a delta that is compatible with the tree structure of XML. However, it is preferable to consider specific algorithms for change detection on tree structures, since they use the knowledge of the structure of the document to improve their efficiency and the quality of their results.

Kuo-Chung Tai [99] gave a definition of the edit distance between ordered labeled tree and the first non-exponential algorithm to compute it. The insert and delete operations are in the spirit of the operations on strings: deleting a node means making its children become children of the node's parent. Inserting is the complement of deleting. Given two documents $D1$ and $D2$, the resulting algorithm has a complexity of $O(|D1| * |D2| * depth(D1)^2 * depth(D2)^2)$ in time and space. Lu's algorithm [68] uses another edit based distance. The idea underlying this algorithm is, when a node in subtree $D1$ matches with a node in subtree $D2$, to use the string edit algorithm to match their respective children.

In Selkow's variant [97], insertion and deletion are restricted to the leaves of the tree. Thus, applying Lu's algorithm in the case of Selkow's variant results in a time complexity of $O(|D1| * |D2|)$. Depending on the considered tree data, this definition may be more accurate. It is used for example, in Yang's [121] algorithm to find the syntactic differences between two programs. Due to XML structure, it is clear that the definition is also accurate for XML documents. An XML Document structure may be defined by a DTD, so inserting and deleting a node and changing its children level would change the document's structure and may not be possible. However, inserting and deleting leaves or subtrees happens quite often, because it corresponds to adding or removing objects descriptions, e.g. like adding or removing people in an address book.

Recently, Sun released an XML specific tool named *DiffMK* [78] that computes the difference between two XML documents. This tool is based on the Unix standard *diff* algorithm, and uses a *list* description of the XML document, thus losing the benefit of the tree structure of XML.

We do not consider here the unordered tree problem [125, 98] nor the tree alignment [60] problems.

Perhaps the closest in spirit to our algorithm is *LaDiff* or *MH-Diff* [25, 24]. It is also designed for XML documents. It introduces a matching criteria to compare nodes, and the overall matching between both versions of the document is decided on this base. The faster version of the matching algorithm uses longest common subsequence computations for every element node starting from the leaves of the document. Its cost is in $O(n * e + e^2)$ where n is the total number of leaf nodes, and e a weighted edit distance between the two trees. More precisely, e is the sum of the number of deleted and inserted subtrees, and the total size of subtrees that moved for the shortest edit script.

Then an edit script conforming to the given matching is constructed in a cost of $O(n * d)$ where n is the total number of nodes, and d the total number of children moving within the same parent. Like most other algorithms, the worst case cost, obtained here considering that large subtrees have moved, is quadratic in the size of the data.

The main reason why few *diff* algorithm supporting *move* operations have been developed earlier is that most formulations of the tree diff problem are NP-hard [126, 24] (by reduction from the ’exact cover by three-sets’). *MH-Diff*, presented in [24] provides an efficient heuristic solution based on transforming the problem to the edge cover problem, with a worst case cost in in $O(n^2 * log(n))$.

Our algorithm is in the spirit of Selkow’s variant, and resembles Lu’s algorithm. The differences come from the use of the structure of XML documents. In Lu’s algorithm, once a node is matched, we try to match its children using the string algorithm. For this, children are identified using their label. But this would not apply in practice on XML documents, as many nodes may have the same label. So we use a signature computed over the children’s subtree. But then, children may not be matched only because of a slight difference in their subtree, so we had to extend our algorithm by taking into consideration those children and their subtree and matching part of it if possible.

Using this edit definition, we could add the support of move operations. Note that a move operation can be seen as the succession of a deletion and an insertion. However it is different in that we consider the cost of $move$ to be much less than the sum of deleting and inserting the subtree. Thus it is clear that previous algorithm wouldn't compute the minimal edit script as we defined it.

Last but not least, our algorithm goal is slightly different from previous algorithms in that for performance reasons, we do not necessarily want to compute the very minimal edit script.

3.4 Brief overview of the change representation model

In this section, we present some aspects of the change model [69] that we use in the present chapter. The presentation will be very brief and omit many aspects of the complete model. A complete representation is given in [69] and in Chapter 4.

The simple model for XML data we consider roughly consists of ordered trees (each node may have a *list* of children) [3]. Nodes also have values (data for text nodes and label for element nodes). We will briefly mention later some specific treatment for attributes. The starting point for the change model is a sequence of snapshots of some XML data. A delta is an XML document that represents the changes between two consecutive snapshot versions of an XML document. It uses persistent node identifiers, namely XIDs, in a critical way. We consider next the persistent identification of XML nodes, and then the *deltas*, a novel representation of changes in XML documents.

Persistent identification of nodes The persistent identification of nodes is the basis of the change representation for XML documents we use. Persistent identifiers can be used to easily track parts of an XML document through time. We start by assigning to every node of the first version of an XML document a unique identifier, for example its postfix position. When a new version of the document arrives, we use the *diff* algorithm to match nodes between the two versions. As previously reported, matched nodes in the new document thereby obtain their (persistent) identifiers from their matching in the previous version. New persistent identifiers are assigned to unmatched nodes. Given a set of matchings between two versions of an XML document, there are only few *deltas* that can describe the corresponding changes. The differences between these *deltas* essentially come from move operations that reorder a subsequence of child nodes for a given parent [69]. More details on the definition and storage of our persistent identifiers, that we call XIDs, are given in [69]. The XID-map is a string attached to some XML subtree that describes the XIDs of its nodes.

Representing changes The delta is a *set* of the following elementary operations: (i) the deletion of subtrees; (ii) the insertion of subtrees; (iii) an update of the value of a text node or an attribute; and (iv) a move of a node or a part of a subtree. Note that it is a *set* of operations. Positions in operations are always referring to positions in the source or target document. For instance, *move*(m, n, o, p, q) specifies that node o is moved from being the n-th child of node m to being the q-th child of p. The management of positions greatly complicates the issue comparing to, say, changes in relational systems. Note also that the model of change we use relies heavily on the persistent identification of XML nodes. It is based on "completed" deltas that contain redundant information. For instance, in case of updates, we store the old and new value. Indeed, a delta specifies both the transformation from the old to the new version, but the inverse transformation as well. Nice mathematical and practical properties of completed deltas are shown in [69]. In particular, we can reconstruct any version of the document given another version and the corresponding delta, and we can aggregate and inverse *deltas*. Finally, observe that the fact that we consider *move* operations is a key difference with most previous work. Not only is it necessary in an XML context to deal with permutations of the children of a node (a frequently occurring situation) but also to handle more general moves as well. Moves are important to detect from a semantic viewpoint. For example consider the XML document in Figure 3.2 (first version) and Figure 3.3 (second version).

Its tree representation is given in the left part of Figure 3.4. When the document changes, Figure 3.4 shows how we identify the subtrees of the new version to subtrees in the previous version of the document. This identification is the main goal of the *diff* algorithm we present here. Once nodes from the two versions have been matched, it is possible to produce a delta. The main difficulty, shown in Section 3.5, is to manage positions. Assuming some identification of nodes in the

```
<Category>
  <Title>Digital Cameras</Title>
  <Discount>
    <Product>
      <Name>  tx123 </Name>
      <Price> $499  </Price>
    </Product>
  </Discount>
  <NewProducts>
    <Product>
      <Name>  zy456 </Name>
      <Price> $799  </Price>
    </Product>
  </NewProducts>
</Category>
```

Figure 3.2: XML Document Example (first version)

```
<Category>
  <Title>Digital Cameras</Title>
  <Discount>
    <Product>
      <Name>  zy456 </Name>
      <Price> $699  </Price>
    </Product>
  </Discount>
  <NewProducts>
    <Product>
      <Name>  abc   </Name>
      <Price> $899  </Price>
    </Product>
  </NewProducts>
</Category>
```

Figure 3.3: XML Document Example (second version)

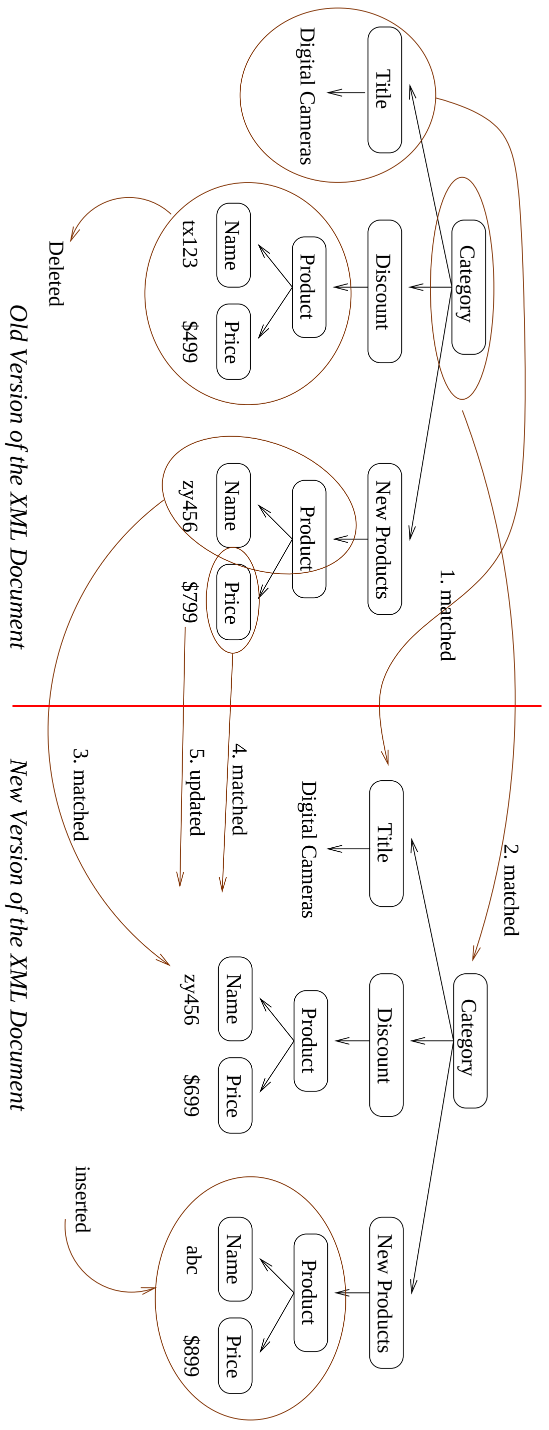

Figure 3.4: Matching subtrees

```
<delete XID="7" XID-map="(3-7)" parentXID="8" pos="1">
  <Product>
    <Name>  tx123 </Name>
    <Price> $499  </Price>
  </Product>
</delete>

<insert XID="20" XID-map="(16-20)" parentXID="14" pos="1">
  <Product>
    <Name>  abc  </Name>
    <Price> $899 </Price>
  </Product>
</insert>

<move XID=13 fromParent="14" fromPos="1"
               toParent= "8" toPos  ="1" />

<update XID="11">
  <old-value>
    $799
  </old-value>
  <new-value>
    $699
  </new-value>
</update>
```

Figure 3.5: XML Delta Example

old version (namely postfix order in the example), the delta representing changes from the old version to the new one may as in Figure 3.5.

It is not easy to evaluate the quality of a *diff*. Indeed, in our context, different usages of the *diff* may use different criteria. Typical criteria could be the size of the delta or the number of operations in it. Choices in the design of our algorithm or in its tuning may result in different deltas, and so different interpretations of the changes that happened between two versions.

3.5 The XyDiff Algorithm

In this section, we introduce a novel algorithm that computes the difference between two XML documents. Its use is mainly to match nodes from the two documents and construct a delta that represents the changes between them. We provide a cost analysis for this algorithm. A comparison with previous work is given in Section 3.3. Intuitively, our algorithm finds matchings between common large subtrees of the two documents and propagate these matchings. XyDiff uses both Bottom-Up and Top-Down propagation of matchings. Matchings are propagated bottom-up and (most of the time), but only lazily down. This approach was preferred to other approaches we considered because it allows to compute the *diff* in linear time. We next give some intuition, then a more detailed description of our algorithm.

3.5.1 Intuition

To illustrate the algorithm, suppose we are computing the changes between XML document $D1$ and XML document $D2$, $D2$ being the most recent version.

The starting point of the algorithm is to match the largest identical parts of both documents. So we start by registering in a map a unique signature (e.g. a hash value) for every subtree of the old document $D1$. If ID attributes are defined in the DTD, we will match corresponding nodes according to their value, and propagate these matching in a simple bottom-up and top-down pass.

Then we consider every subtree in $D2$, starting from the largest, and try to find whether it is identical to some of the registered subtrees of $D1$. If so, we match both subtrees. (This results in matching every node of the subtree in $D1$ with the respective node of the subtree in $D2$.) For example, in Figure 3.4, we do not find an identical subtree for the tree starting at *Category*, but the subtree starting at *Title* is matched.

We can then attempt to match the parents of two matched subtrees. We do that only if they have the same labels. Clearly, there is a risk of forcing wrong matches by doing so. Thus, we control the propagation of a matching bottom-up based on the length of the path to the ancestor and the weight of the matching subtrees. For example, a large subtree may force the matching of its ancestors up to the root, whereas matching a small subtree may not even force the matching of its parent.

The fact that the parents have been matched may then help detect matchings between descendants because pairs of such subtrees are considered as good *candidates* for a match. The matching of large identical subtrees may thus help matching siblings subtrees which are slightly different. To see an example, consider Figure 3.4. The subtree *Name/zy456* is matched. Then its parent *Product* is matched too. The parents being matched, the *Price* nodes may eventually be matched, although the subtrees are different. (This will allow detecting that the price was updated.) When both parents have a single child with a given label, we propagate the match immediately. (It is possible to use data structures that allow detecting such situations at little cost.) Otherwise, we do not propagate the matching immediately (lazy down). Future matchings (of smaller subtrees) may eventually result in matching them at little cost.

The lazy propagation downward of our algorithm is an important distinction from previous work on the topic. Note that if the two matched nodes have m and m' children with the same label ℓ, we have $m \times m'$ pairs to consider. Attempting this comparison on the spot would result in a quadratic computation.

We start by considering the largest subtrees in $D2$. The first matchings are clear, because it is very unlikely that there is more than one large subtree in $D1$ with the same signature. However it is often the case that when the algorithm goes on and considers smaller subtrees, more than one subtrees of $D1$ are identical to it. We say then that these subtrees are candidates to matching the considered subtree of $D2$. At this point, we use the precedent matches to determines the best candidate among them, by determining which is closest to the existing matches. Typically, if one of the candidate has its parent already matched to the parent of the considered node, it is certainly the best candidate. And thanks to the order in which nodes are considered, the position among siblings plays an important role too.

When this part of the algorithm is over, we have considered and perhaps matched every node of $D2$. There are two reasons why a node would have no matching: either because it represents new data that has been inserted in the document, or because we missed matching it. The reason why the algorithm failed may be that at the time the node was considered, there was no sufficient knowledge or reasons to allow a match with one of its candidates. But based on the more complete knowledge that we have now, we can do a "peephole" optimization pass to retry some of the rejected nodes. Aspects on this bottom-up and top-down simple pass are considered in Section 3.5.3.

In Figure 3.4, the nodes *Discount* has not been matched yet because the content of its subtrees has completely changed. But in the optimization phase, we see that it is the only subtree of node *Category* with this label, so we match it.

Once no more matchings can be obtained, unmatched nodes in $D2$ (resp. $D1$) correspond to inserted (resp. deleted) nodes. For instance, in Figure 3.4, the subtrees for products *tx123* and *abc* could not be matched and so are respectively considered as deleted and inserted data. Finally, a computationally non negligible task is to consider each matching node and decide if the node is at its right place, or whether it has been moved.

3.5.2 Detailed description

The various phases of our algorithm are detailed next.

Phase 1 (Use ID attributes information): In one traversal of each tree, we register nodes that are uniquely identified by an ID attribute defined in the DTD of the documents. The existence of ID attribute for a given node provides a unique condition to match the node: its matching must have the same ID value. If such a pair of nodes is found in the other document, they are matched. Other nodes with ID attributes can not be matched, even during the next phases. Then, a simple bottom-up and top-down propagation pass is applied. Note that if ID attributes are frequently used in the documents, most of the matching decision have been made during this phase.

Phase 2 (Compute signatures and order subtrees by weight): In one traversal of each tree, we compute the signature of each node of the old and new documents. The signature is a hash value computed using the node's content, and its children signatures. Thus it uniquely represents the content of the entire subtree rooted at that node. A weight is computed simultaneously for each node. It is the size of the content for text nodes and the sum of the weights of children for element nodes.

We construct a priority queue designed to contain subtrees from the new document. The subtrees are represented by their roots, and the priority is given by the weights. The queue is used to provide us with the next heaviest subtree for which we want to find a match. (When several nodes have the same weight, the first subtree inserted in the queue is chosen.) To start, the queue only contains the root of the entire new document.

Phase 3 (Try to find matchings starting from heaviest nodes): We remove the heaviest subtree of the queue, e.g. a node in the new document, and construct a list of *candidates,* e.g. nodes in the old document that have the same signature. From these, we get the best candidate (see later), and match both nodes. If there is no matching and the node is an element, its children are added to the queue. If there are many candidates, the best candidate is one whose parent matches the

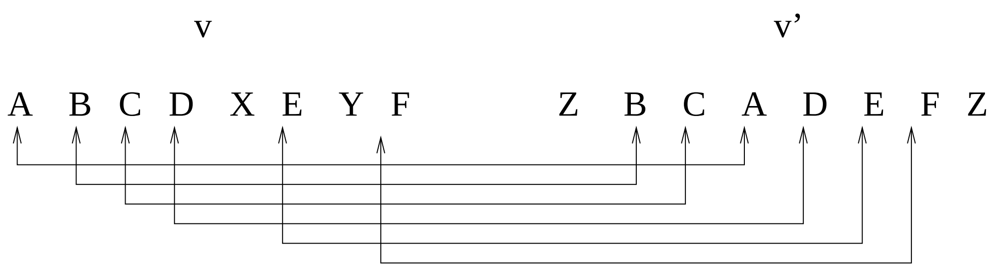

Figure 3.6: Local moves

reference node’s parent, if any. If no candidate is accepted, we look one level higher. The number of levels we accept to consider depends on the node weight.

When a candidate is accepted, we match the pair of subtrees and their ancestors as long as they have the same label. The number of ancestors that we match depends on the node weight.

Phase 4 (Optimization: Use structure to propagate matchings): We traverse the tree bottom-up and then top-down and try to match nodes from the old and new documents such that their parents are matching and they have the same label. This propagation pass significantly improves the quality of the delta and more precisely avoids detecting unnecessary insertions and deletions. The main issue of this part is to avoid expensive computations, so specific choices are explained in Section 3.5.3.

Phase 5 (Compute the delta): This last phase can itself be split in 3 steps:

1. Inserts/Deletes/Updates: Find all unmatched nodes in the old/new document, mark them as *deleted/inserted*; record the effect of their deletion/insertion to the position of their siblings. If a text node is matched but its content has changed, we will mark it as *updated*.

2. Moves: Find all nodes that are matched but with non matching parents. These correspond to *moves*. Nodes that have the same parent in the new document as in the old document may have been moved within these parents. This is discussed further.

3. These operations are reorganized and the delta is produced. (Details omitted.)

Remark. Let us now consider the issue of *moves* within the same parents. For this, consider the example in Figure 3.6. Two nodes v (in the old version) and v' (in the new version) have been matched. There may have been deletions, and also moves from v to some other part of the document, e.g. X and Y are in v but not in v'. Conversely, there may be insertions and moves from some part of the document to v', e.g. two Z nodes are in v' but not in v. The other nodes are pairs of nodes, where one node is in v, and its matching node in v'. For instance, the A node in v matches the A node in v'.

However, they need not be in the same order. When they are not, we need to introduce more moves to capture the changes. In Figure 3.6, the lines represent matchings. There are 6 pairs of matched nodes in v and v', corresponding to the two sequences A, B, C, D, E, F (in v) and B, C, A, D, E, F (in v').

To compute a minimum number of moves that are needed, it suffices to find a (not necessarily unique) *largest order preserving subsequence*. Here such a sequence is B, C, D, E, F in v that matches B, C, D, E, F in v' while preserving the order. Then we need only to add move operations for the other pair of nodes, here a single operation is sufficient: $move(v/Atov'/A)$. In XyDiff, we use a more general definition and algorithm where the cost of each move corresponds to some weight assigned to each node. This gives us the “minimum” set of moves.

However, finding the largest order preserving subsequence is expensive for large sequences. More precisely the time and space cost is quadratic in the number of nodes (see Section 3.3). Thus, for performance reasons, we use a heuristic which does not guarantee optimality, but is faster and proves to be sufficient in practice. It is used when the number of children is large, and it works by cutting it into smaller subsequences with a fixed maximum length (e.g. 50). We apply on them the longest common subsequence algorithms(see Section 3.5.3), and merge the resulting subsequences. The result is a subquence that is clearly a common subsequence of the two original lists of children, although in general not the longest one.

For instance, consider the two sequences $A, B, C, D, E, F, G, H, I, J$ and $D, E, J, I, A, B, C, F, G, H$. The two sequences are too large, and we apply the quadratic subsequence algorithm only to their left and right half. More precisely, we find the longest common subsequence of A, B, C, D, E and D, E, J, I, A, which is D, E. Then we find the longest common subsequence of F, G, H, I, J and B, C, F, G, H, which is F, G, H. By merging the two results, we obtain D, E, F, G, H (length 5) as a possible solution. However, the longest common subsequence would have been A, B, C, F, G, H (length 6).

Tuning The details of our algorithm require some choices that we describe next. We also consider the tuning of some parameters of the system.

First, we have to select an appropriate definition for weight. The choice of a weight has impact on the accuracy of matches, and therefore both on the quality and speed of the algorithm. We will see in Section 3.5.3, that the weight of an element node must be no less than the sum of its children. It should also grow in $O(n)$ where n is the size of the document. We use $1+sum(weight(children))$. For text nodes (i.e. leaves), we consider that when the text is large (e.g. a long description), it should have more weight than a simple word. We use $1 + log(length(text))$ as a measure.

Also, when matching two subtrees, it is not easy to choose how far to go up in matching ancestor nodes in the hierarchy. A too small distance would result in missing matches whereas a too large one may generate erroneous matches (e.g. matching many ascendant nodes because two insignificant texts are identical). We note the maximum distance (depth) d. We want d to be greater for larger subtrees, i.e. for some node, d grows with the weight W of the corresponding subtree. For performance reasons (time complexity), we show in Section 3.5.3 that, if no specific index is used, the upper bound for d is in $O(log(n) * \frac{W}{W_0})$ where W is the weight of the corresponding subtree, and W_0 the weight for the whole document. However, this implies in general that for a given subtree with weight W, the distance d decreases to zero when the document becomes larger (i.e. W_0 goes to infinity). In 3.5.3 we explain how to use indexes to enable greater values of d. More precisely, we use $d = 1 + \frac{W}{W_0}$ in XyDiff, where h is the lower value between $log(n)$ and the depth of the root the subtree starting from the root of the document. The $+1$ value means that we use a level-1 index (see Section 3.5.3).

Other XML features We briefly mention here two other specific aspects of XML that have impacts on the *diff*, namely attributes and DTDs.

First, consider attributes. Attributes in XML are different from element nodes in some aspects. First, a node may have at most one attribute of label ℓ for a given ℓ. Also, the ordering for attributes is irrelevant. For these reasons, we do not provide persistent identifiers to attributes, i.e., a particular attribute node is identified by the persistent identifier of its parent and its label (so in our representation of *delta*, we use specific update operations for attributes). When two elements are matched between two consecutive versions, the attributes with the same label are automatically matched. We also use ID attributes (XML identifiers) to know if the nodes owning these attributes should (or can't) be matched as well.

Now consider DTDs, a most important property of XML documents that allows to type them. We have considered using this information to improve our algorithm. For instance, it may seem useful to use the information that an element of label ℓ has at most one child of label ℓ' to perform matching propagation. Such reasoning is costly because it involves the DTD and turns out not to help much because we can sometimes obtain this information at little cost on the document itself, even when the DTD does not specify it. On the other hand, the DTD or XMLSchema (or a data guide in absence of DTD) is an excellent structure to record statistical information. It is therefore a useful tool to introduce learning features in the algorithm, e.g. learn that a price node is more likely to change than a description node. This was not used in our implementation.

3.5.3 Complexity analysis

In this section, we determine an upper bound for the cost of our algorithm, and we explain the use of indexes in a critical part of the algorithm. For space reasons, we do not present the algorithmic of the different functions here.

Note that the number of nodes is always smaller than n where n is the size of both document files.

Matching Nodes. First, reading both documents, computing the hash value for signatures, and registering ID attributes in a hash table is linear in time and space. The simple bottom-up and top-down pass -used in the first and fourth phase- works by considering some specific optimization possibilities on each node. These passes are designed to avoid costly tests. They focus on a fixed set of features that have a constant time and space cost for each (child) node, so that their overall cost is linear in time and space:

1. propagate to parent: Consider that node i is not matched. If it has a children c matched to some node c' we will match i to the parent i' of c'. If i has many matched children $c1, c2, ...$, then there are many possibilities for i'. So we will prefer the parent i' of the larger (weight) set of children $c'1, c'2, ...$. The computation is done in postfix order with a tree traversal.

2. propagate to children: If a node is matched, and both it and its matching have a unique children with a given label, then these two children will be matched. Again the cost is no more than of a tree traversal.

During the XyDiff algorithm, the worst-case occurs when no node is matched. In this case, every node is placed into the priority queue, with an inserting cost of $log(n)$ (ordered heap). This results in a total upper bound of $n * log(n)$. The memory usage is linear in the size of the documents.

For every node, a call is made to the function that finds the best candidate (the set of candidates is obtained using a hash table created in the first phase). The general definition of this function would be to enumerate all candidates and choose the best one as the one with the closest ascendant. It works by enumerating candidates and testing the ascendant up to a given depth. Thus the time cost is in $O(c * d)$ where c is the number of candidates, and d the maximum path length allowed for ancestor's look-up. As previously described, we make d depend on the weight W of the subtree. Thanks to the first rules defined in previous section, and because identical subtrees can not overlap, c is smaller than W_0/W where W_0 is the weight for the subtree representing the whole document. The second rule states that $d = O(log(n) * W/W_0)$. So the cost of a function call is in $O(log(n))$. The overall cost is then in $O(n * log(n))$.

Indexes. However, this upper limit for d means that when the document's size increases and W_0 goes to infinity, d goes to zero. This implies that it would not be possible to test all candidates. The issue occurs, for instance, when there are multiple occurrences of some text node in a large document, e.g. a company name, an email address or the URL of some web site. We wish to be able to test each candidate at least once. In other words, d should be greater than 1. We use for instance $d = 1 + h * \frac{W}{W_0}$. To do so, a specific index (a hash table) is created during initialization. The basic index retrieves all candidate nodes for a given signature. This specific index retrieves all candidate nodes for a given signature and a parent node identifier. This is equivalent to using the basic index for finding all candidates with that signature, and then test each of them to find the ones with the proper parent node (i.e. $d = 1$). In other words, the best candidate (if any) is found in constant time. This generic solution works for any lower bound of d by using as many indexes to access nodes by their grand-parent or ascendant identifier. The extra cost is to construct these indexes during initialization. If D is the lower bound for d, the cost of constructing the index is in $O(D * n)$, where n is the size of the documents.

Delta Construction. The second part consists of constructing the *delta* using the matchings obtained previously. Finding nodes that have been deleted or inserted only requires to test if nodes of both documents have been matched. It is also clear that a node has moved if its parent and the parent of its matching do not match. So this first step is linear in time and space. The difficulty comes with nodes that stay within the same parent. If their order has changed, it means that some of them have 'moved'. As mentioned above, to obtain the optimal *delta*, we should apply a 'longest common subsequence' algorithm on this sequence of children [69]. These algorithms have typically a time cost of $O(s^2/log(s))$, where s is the number of children, and a space cost of $O(s^2)$. However, in practical applications, applying this algorithm on a fixed-length set of children (e.g. 50), and merging the obtained subsequences, provides excellent results and has a time and space cost in $O(s)$. We choose this heuristic, so the total cost for the document is then in $O(n)$.

So the overall worst-case cost is $O(n * (log(n))$ where n is the size of the document files (including the DTD, if any, that we also have to read). The memory usage is linear in the total size of both documents.

3.6 Experiments

In this section we present an experimental study of the algorithm. We show that it achieves its goals, in that it runs in linear time, and computes good quality deltas. (The linear space bound is obvious and will not be discussed.) We first present results on some synthesized data (synthetic changes on XML documents). We then briefly consider changes observed on the web. Due to space limitations only a small portion of the experiments will be presented here. However, they illustrate reasonably well what we learned from the experiments. More experiments are presented in Chapter 5.

3.6.1 Measures on simulated changes

The measures show that the algorithm is very fast, almost linear in the size of data. Also, since it does not guarantee an optimal result, we analyze the quality of its result and show experimentally that it is excellent. For these experiments, we needed large test sets. More precisely, we needed to be able to directly control the changes on a document based on parameters of interest such as deletion rate. To do that, we built a change simulator that we describe next.

Change simulator The change simulator allows to generate changes on some input XML document. Its design is very important as any artifact or deviation in the change simulator may eventually have consequences in the test set. We tried to keep the architecture of the change simulator very simple. The change simulator reads an XML document, and stores its nodes in arrays. Then, based on some parameters (probabilities for each change operations) the four types of simulated operations are created in three phases:

[delete] Given a delete probability, we delete some nodes and its entire subtree.

[update] The remaining text nodes are then updated (with original text data) based on their update probability.

[insert/move] We choose random nodes in the remaining element nodes and insert a child to them, depending on the insert and move probability. The type of the child node (element or text) has to be chosen according to the type of its siblings, e.g. we do not insert a text node next to another text node, or else both data will be merged in the parsing of the resulting document. So according to the type of node inserted, and the move probability we do either insert data that had been deleted, e.g. that corresponds to a move, or we insert "original" data. For original data, we try to match to the XML style of the document. If the required type is text, we can just insert any original text using counters. But if the required node has to be a tag, we try to copy the tag from one of its siblings, or cousin, or ascendant; this is important for XML document in order to preserve the distribution of labels which is, as we have seen, one of the specificities of XML trees.

Note that because we focused on the structure of data, all probabilities are given *per node*. A slightly different model would be obtained if it was given *per byte of data*. Note also that because the number of nodes after the first phase is less than the original number of nodes of the document, we recompute update and insert probabilities to compensate.

The result of the change simulator is both a delta representing the exact changes that occurred, which will be useful to compare later with the algorithmically computed delta, and a new version of the document. It is not easy to determine whether the change simulator is good or not. But based on statistical knowledge of changes that occurs in the real web (see further), we will be able to evaluate it and tune it. We tried to verify both by human evaluation of resulting documents and by the control of measurable parameters (e.g. size, number of element nodes, size of text nodes, ...) that the change simulator behaves properly. The change simulator we presented here is the result of a few iterations. It seems now to conform reasonably to our expectations.

Performance We verify next that the complexity is no more than the expected $O(n * log(n))$ time. To do that, we use the change simulator to create arbitrary sized data and measure the time needed to compute the *diff* algorithm. In the experiment we report next, the change simulator was set to generate a fair amount of changes in the document, the probabilities for each node to be modified, deleted or have a child subtree inserted, or be moved were set to 10 percent each. Measures have been conducted many times, and using different original XML documents.

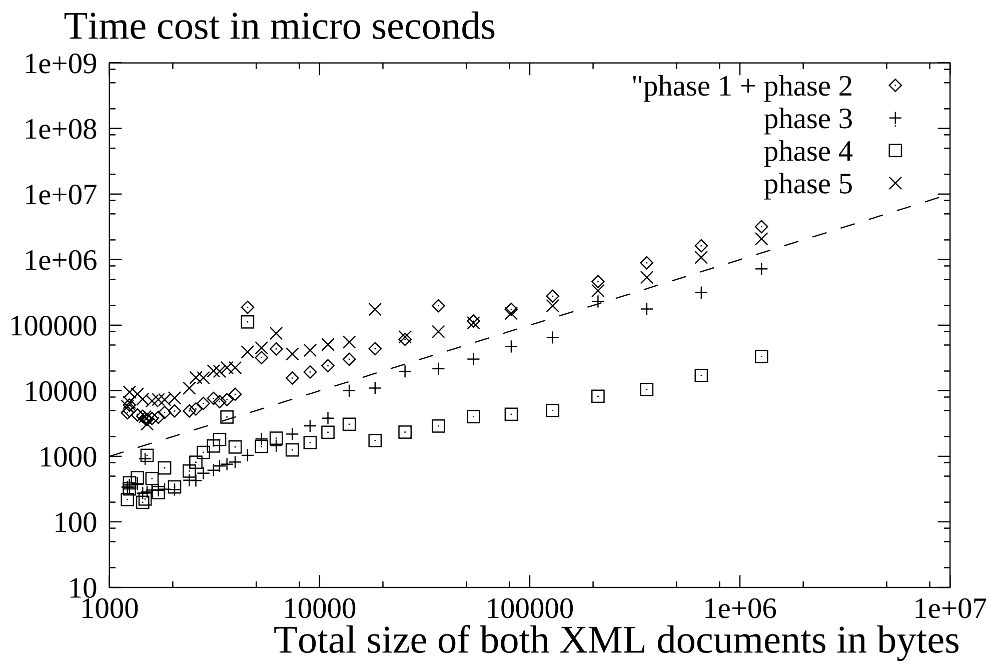

Figure 3.7: Time cost for the different phases

The results (see Figure 3.7) show clearly that the algorithm's cost is almost linear in time[1]. We have analyzed precisely the time spent in every function, but due to lack of space, we do not provide full details here. Phases $3 + 4$, the core of

[1]A few values are dispersed because of the limitations of our profiling tool.

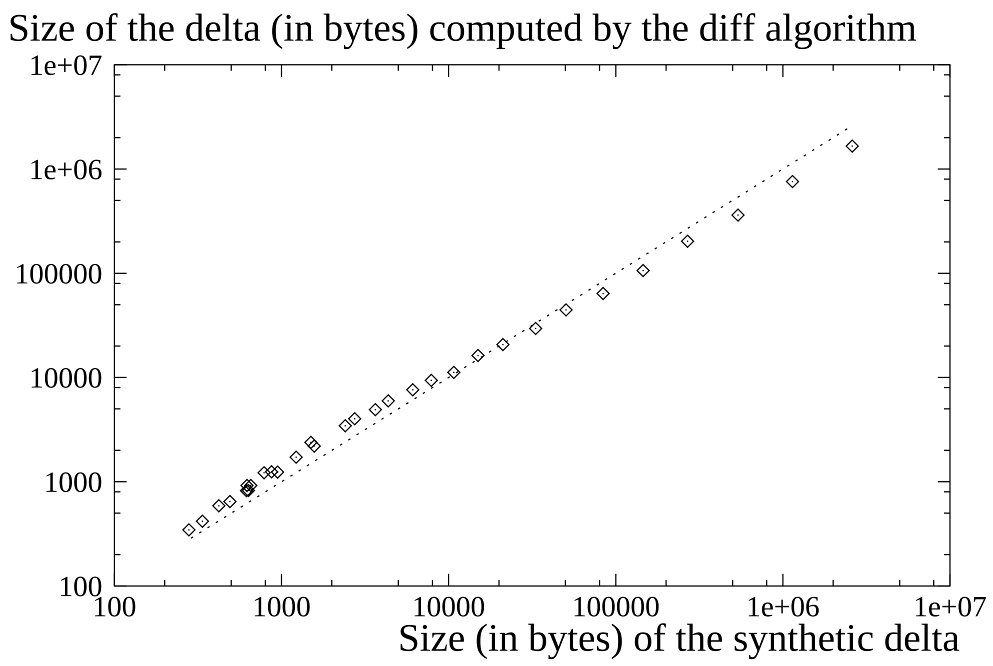

Figure 3.8: Quality of Diff

the *diff* algorithm, are clearly the fastest part of the whole process. Indeed, most of the time is spent in parts that manipulate the XML data structure: (i) in Phase 1 and 2, we parse the file [113] and hash its content; (ii) in Phase 5, we manipulate the DOM tree [113]. The progression is also linear. The graph may seem a bit different but that comes from the fact that the text nodes we insert turn out to be on average smaller than text nodes in the original document.

A fair and extensive comparison with other *diff* programs would require a lot more work and more space to be presented. An in-depth comparison, would have to take into account speed, but also, quality of the result ("optimality"), nature of the result (e.g., moves or not). Also, the comparison of execution time may be biased by many factors such as the implementation language, the XML parser that is used, etc. Different algorithms may perform differently depending on the amount and nature of changes that occurred in the document. For example, our *diff* is typically excellent for few changes.

Quality We analyze next the quality of the *diff* in various situations, e.g. if the document has almost not changed, or if the document changed a lot. We paid particular attention to move operations, because detecting move operations is a main contribution of our algorithm.

Using our change simulator, we generated different amounts of changes for a sequence of documents, including a high proportion of move operations. In Figure 3.8, we compare the size of the delta obtained using XyDiff to the size of the original delta created by the change simulator. The delta obtained by the simulator captures the edit script of operations that has been applied to the original document to change it, and, in that sense, it can be viewed as *perfect*. Delta's sizes are expressed in bytes. The original document size varies from a few hundred bytes, to a megabyte. The average size of an XML document on the Web is about twenty kilobytes. The points in Figure 3.8 are obtained by varying the parameters of the change. Experiments with different documents presented the same patterns.

The experiment shows that the delta produced by *diff* is about the size of the delta produced by the simulator. This is the case even when there are many updates including many move operations. For an average number of changes, when about thirty percent of nodes are modified, the delta computed by the *diff* algorithm is about fifty percent larger. This is precisely due to the large number of move operations that modify the structure of the document. But when the change rate increases further, the delta gains in efficiency again, and is even sometimes more accurate than the original delta, in that it finds ways to compress the set of changes generated by the simulator. Note that the efficiency lost in the middle of the range is very acceptable, because (i) the corresponding change rate is much more than what is generally found on real web documents; and (ii) the presence of many moves operations modifying the structure of the document is rare on real web documents.

3.6.2 Measures on real web data

We mention next results obtained by running our algorithm over more than ten thousands XML documents crawled on the Web [79]. Unfortunately, few XML documents we found changed during the time-frame of the experiment. We believe that it comes from the fact that XML is still in its infancy and XML documents on the web are less likely to change than HTML documents. This is also due to the fact that the time-frame of the experiment was certainly too short. More experiments are presented in Chapter 5.

We present here results obtained on about two hundred XML documents that changed on a per-week basis. This sample is certainly too small for statistics, but its small size allowed a human analysis of the *diff* outputs. Since we do not have here a “perfect” delta as in the case of synthesized changes, we compare our results to Unix Diff. Our test sample also contains about two hundred large XML documents representing metadata about web sites. We also applied the *diff* on a few large XML files (about five megabytes each) representing metadata about the entire INRIA web site.

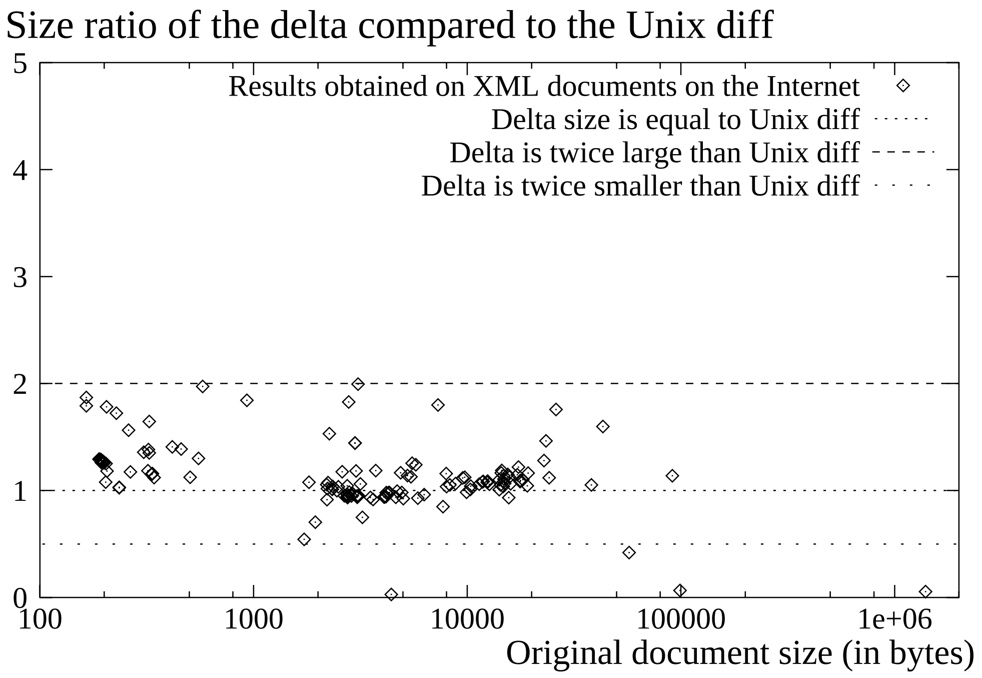

Figure 3.9: Delta over Unix Diff size ratio

The most remarkable property of the *deltas* is that they are on average roughly the size of the Unix Diff result (see Figure 3.9). The outputs of Unix Diff and of our algorithm are both sufficient to reconstruct one version from another, but *deltas* contain a lot of additional information about the structure of changes. It is interesting to note that the cost paid for that extra information is very small in average.

It is also important to compare the delta size to the document’s size, although this is very dependent on how much the document changed. Other experiments we conducted [69] showed that the delta size is usually less than the size of one version. In some cases, in particular for larger documents (e.g. more than 100 kilobytes), the delta size is less than 10 percent of the size of the document.

One reason for the delta to be significantly better in size compared to the Unix Diff is that it detects moves of big subtrees. In practice, this does not occur often.

A drawback of the Unix Diff is that it uses *newline* as separator, and some XML document may contain very long lines.

We have also tested XyDiff on XML documents describing portions of the web, e.g., web sites. We implemented a tool that represents a snapshot of a portion of the web as an XML document. For instance, using the site `www.inria.fr` that is about fourteen thousands pages, the XML document is about five megabytes. Given snapshots of the web site (i.e. given two XML documents), XyDiff computes what has changed in the time interval. XyDiff computes the delta in about thirty seconds. Note that the core of our algorithm is running for less than two seconds whereas the rest of the time is used to read and write the XML data. The *delta*'s we obtain for this particular site are typically of size one megabytes. To conclude this section, we want to stress the fact that although the test set was very small, it was sufficient to validate the formal analysis. More experiments are clearly needed.

3.7 Conclusion

All the ideas described here have been implemented and tested. A recent version of XyDiff can be downloaded at [34]. We showed by comparing our algorithm with existing tree pattern matching algorithms or standard *diff* algorithms, that the use of XML specificities leads to significant improvements.

We already mentioned the need to gather more statistics about the size of deltas and in particular for real web data. To understand changes, we need to also gather statistics on change frequency, patterns of changes in a document, in a web site, etc. Many issues may be further investigated. For example we can extend our use of DTDs to XMLSchema. Other aspects of the actual implementation could be improved for a different trade-off in quality over performance, e.g. we could investigate the benefits of intentionally missing *move* operations for children that stay with the same parent.

Chapter 4

An XML representation of changes in XML documents: XyDelta

Abstract *There are several possible ways to represent the change information in order to build a temporal XML data warehouse. One of them is to store, for each document, some* deltas *that represent changes between versions of the documents. It is then possible to issue queries on the deltas, in particular if they are themselves XML documents. In this chapter, we present our work on the topic of representing XML changes in XML, and more precisely we detail some formal aspects of our representation, namely* XyDelta.

This work was performed with Amélie Marian, Serge Abiteboul and Laurent Mignet. An article has been published in [69]. The project was originally started by Amélie Marian before she left for Columbia University. I was then leading the work. Marian worked on the definition of persistent identifiers (XIDs), and the XML delta format. She also developed the first prototype of algorithms to apply, revert and aggregate deltas. Finally, she conducted experiments to evaluate the effectiveness of deltas for storing versions.

My contributions to that work are:

- *The formal definition of set-based deltas as opposed to edit-scripts (see below), and the notion of "equivalent" deltas.*

- *Linear time algorithms (and their implementation) to apply deltas, invert them, and aggregate them. The core of this work consists in the definition of an order relationship between nodes that is used to order operations.*

- *The formal (re)definition of move operations.*

This section presents my contributions. They have been clearly influenced by the original work of Marian. Note that the XyDiff output discussed in previous chapter is in XyDelta.

4.1 Introduction

In previous chapter, we explained how to detect changes between two versions of an XML document. One possible use of change information is to use matching between nodes and provide a persistent identification mecanism. This is done, for instance, by storing a “large” version of the document that contains an aggregation of all past data fragments, with annotations that indicates during which period of time each data fragment was present in the document. This approach is efficient for databases where reliable identification information is available for each piece of data, e.g. scientific data with keys [19, 18].

Another possible use of change information is to store, for each document, some *deltas*, that represent changes between versions of the documents. This approach is often prefered to the previous one when the only information available are snapshot versions of the documents. A famous example is CVS [39] that uses deltas to store versions of program source files. In this chapter, we consider this approach that we chose in the context of the Xyleme project [117] where documents were retrieved from the Web. More precisely, we consider the use of XML deltas to represent changes in XML documents. To analyze changes, it is then possible to issue queries on the deltas, since they are themselves XML documents.

Currently, there is no accepted standard on this topic, although several proposals have been made since we introduced our deltas, in particular XUpdate [114], DeltaXML [44] and Microsoft XDL [76]. In the next chapter, these proposals are compared to our representation.

Deltas. Consider snapshot versions of an XML document at some time t. The changes between the snapshot at time t, and the snapshot at time $t+1$, form a delta $delta_{t,t+1}$.

A delta $delta_{i,j}$ represents all changes that occurred between the snapshot of the database (or document) at time i, and the snapshot at time j. Such a delta consists in general in change operations (e.g. insert, delete, update), that describe a way of transforming version i of the document into a version j. If $i < j$, the delta is called a forward delta. If $i > j$, the delta is called a backward delta.

Most previous works uses deltas that are ordered sequences of operations. More precisely, each delta $delta_{i,j}$ consists in a sequence of fine grain operations $Op_1, Op_2, Op_3, ..., Op_n$. These operations are applied to the document version i in the sequence order. The consistency of the delta is its ability to transform a document version i into some document version j. It is possible that one operation inserts a node in an XML document, and another one inserts some other node below it. The second operation is meaningless in absence of the first one.

Motivations. One of our goals was to consider, if possible, the atomic operations as independent one from an other. There are many advantages to such an independance:

- Improve efficiency. It is often the case that a large number of changes occurred between two versions of a document. It is inefficient to apply them to a document if one has to consider changes only in a specific ordering given by the sequence. The ability to analyze groups of them separately is also useful to monitor specific changes.

- Concurrency control. Management of independent changes gives more flexibility for concurrency control, e.g. to manage updates by different users on the same document.

- Semantics of changes. Each independent operation should have a precise semantic. On the opposite, a drawback of editing scripts is that a valid edit script may contain operations that have little semantics. For instance, it may insert a node in the document, and later delete that node, or change several time the same text node.

- Comparing changes. The same changes may be represented by several possible edit scripts. Our model makes comparison of deltas more efficient.

Storage Strategies. Note that when deltas are used, several storage policies can be chosen [69, 29]. For instance, storing the latest version of the database, and all $delta_{t,t-1}$ backward deltas in order to be able to reconstruct any version of the database. Another possibility is to store the first version of the database, say at $t = 0$, and all forward deltas $delta_{0,t}$. It is also possible to store only snapshot versions of the database, and construct the deltas on the fly when necessary. This issue is ignored here.

Organization. In Section 4.2, we first propose a formal definition for edit scripts on XML documents, and we describe some properties of edit scripts. Then we introduce XyScripts, which are specific edit scripts, and XyDelta, representing a class of equivalent edit scripts. Finally, we show that each edit script can be transformed into "its" XyDelta. In Section 4.3, we show how our model can be extended to support some specific XML concepts, and other editing operations such as *update* and *move*. The last section is a conclusion.

4.2 From Edit-Scripts to XyDelta

In this section, we formally define XyDelta based on edit scripts for XML documents. We only consider two possible operations: insert and delete. Others operations (update, move) are discussed in the next section.

In this section, we consider two version i and j of an XML document. We suppose that, in the first version of the document (i.e. i), each node can be uniquely identified by an identifier named its XID. This means that each node of the first document is tagged with an XID, as proposed by Marian and al. in [69].

Note that the implementation is in fact different than adding a tag to each node. We ignore here some subtlety of the management of XID as proposed by Marian and al. We only assume that an XID-Map is attached (virtually) to the document. An XID-Map is a string representation of sequence of XID. When an XID-Map is (virtually) attached to some XML sutree, it provides a persistent identifier to all the nodes in the subtree.

4.2.1 A definition for an Edit-Script

For clarity reasons, we first consider only one type of node, e.g. element nodes. In particular, we ignore attributes and text nodes. Extending the model to support attributes and text nodes is straightforward and is briefly considered further.

We also ignore the problem of managing white-spaces in XML documents. White-spaces are used in XML documents mainly for textual readability, but their management may become a technical issue since XML tools and models (e.g. DOM, SAX, XPath) give them different semantics.

We first define *delete* and *insert* operations, then we define edit scripts and some of their properties.

Definition 4.2.1 (delete)
A delete operation consists in deleting some node n and the entire subtree rooted at that node from a document. The delete operation descriptor contains:

- The deleted subtree (starting at some node n), and its XID-Map (including the XID of n).
- The XID of the parent node p of n.
- The position of n in the ordered sequence of children of p.

Consider a description of some *delete* operation on some document D. In order for the *delete* operation to be valid with D, the node n must exists, its parent p also, the position has to be correct and the subtree content listed with the operation has to be identical (including XIDs) to the subtree contained in the document.

An example of delete is as follows:

```
<delete
    parentXID="18"
    position="2"
    DataXIDmap="(12-13)" >

    <Tag>
         This data is deleted,
         including the tag "Tag"
         <subtag>subtext</subtag>
    </Tag>

</delete>
```

Definition 4.2.2 (insert)
An insert operation consists in inserting an XML subtree in some document. The insert operation descriptor contains:

- The inserted subtree (starting at root n), and its XID-Map (including the XID of n).

- The XID of the parent node p where the subtree should be inserted.
- The position of n in the ordered sequence of children of p.

Thus, an *insert* operation is exactly symmetrical to an *delete* operation. Again, consider some document D and an insert operation. For the operation to be valid with D, the parent p must exists, the position must be valid (i.e. between 1 and the number of children of p plus 1), and the XID-Map of the inserted subtree should contain no XID value that is used in some other place of the document. A typical example is:

```
<insert
    parentXID="18"
    position="2"
    DataXIDmap="(12-13)" >

    <Tag>
         This data is deleted,
         including the tag "Tag"
         <subtag>subtext</subtag>
    </Tag>

</insert>
```

It is important to note that the definition of *insert* and *delete* are symmetrical. More precisely, consider some document D on which a delete operation may be applied that results in document D'. Then, the insert operation obtained by renaming *delete* into *insert* and using the exact same attributes transforms D' into D.

Definition 4.2.3 (Edit-Script)
An edit script is an ordered sequence of delete and insert operations. Let an edit script S be defined by $Op_1, Op_2, ..., Op_n$. Let D_0 be exactly the document D. S is valid *with D if:*

- *Op_i is consistent with D_{i-1}*
- *D_i is the document resulting when applying Op_i to D_{i-1}*

From here, when we consider some edit script, we always mean a valid edit script. We also note $S(D)$ the result document, i.e. D_n.

Definition 4.2.4 (Equivalence)
Two edit scripts S and S' are equivalent *iff: (i) for each D, S is valid for S iff S' is valid for D, (ii) for each D valid with S and S', $S(D) = S'(D)$*

Note that the notion of equality between documents is here the XML content identity as defined in the XML standard [102]. For simplicity, we also use a weak notion of equivalence: S and S' are equivalent *for D*, if both are valid with D, and $S(D) = S(D')$. For instance, the following edit script is equivalent (for any document containing a node with XID 7; and no 99 node), to the empty script:

```
<insert DataXIDmap="99" parentXID="7" position="1">
    <testTag/>
</insert>
<delete DataXIDmap="99" parentXID="7" position="1">
    <testTag/>
</insert>
```

Definition 4.2.5 (Aggregation/Composition)
We define the aggregation of two edit scripts S and S' as the edit script corresponding to the concatenation of their two sequences of operations. It is noted $S'(S)$.

If S is valid with some document D, and S' is valid with $S(D)$, then $S'(S)$ is valid with D.

Aggregation of edit scripts shows the drawback of this classical notion of edit scripts. Consider for instance the previous example. S consists in the first insert operation, and S' in the delete operation that follows and deletes the inserted node. For any document D such that $S'(S)$ is valid with D, we can say that $S'(S)$ is equivalent to the empty script. However, strictly speaking, $S'(S)$ is not equivalent to the empty script since there are some documents on which $S'(S)$ could not be applied. With the model that we introduce next, we focus on equivalent edit scripts by considering only the effect on their source and target document D and $S(D)$.

4.2.2 Changing the order of operations in edit scripts

We present here a set of simple *swap* operations that transform some edit script in an equivalent edit script by swapping two consecutive operations. By making possible the swap of consecutive operations, we enable the complete reordering of any edit script. This is the basis of the XyDelta model. More precisely, we will be swapping to rewrite edit scripts into some ”normal form” with some nice properties. We present a series of lemma that allow to handle the various cases that may arise. The proofs are straightforward, so omitted.

Lemma 4.2.6
Let S be an edit script containing two consecutive *delete* operations $X1$ and $X2$ with the same parent node. Let $p1$ and $p2$ be the respective node's positions. If $p1 \leq p2$, then S' obtained by swapping the two operations, and replacing $p2$ by $p2 + 1$, is equivalent to S.

The goal is that position of delete operations should refer to the position of the nodes before the operations are executed. An example is as follows. Consider a node 1, with four child nodes $101, 102, 103$ and 104. The edit script:

```
...
<delete parentXID="1" DataXIDmap="102" position="2">
   <child2 />
</delete>
<delete parentXID="1" DataXIDmap="104" position="3">
   <child4 />
</delete>
...
```

may be transformed into the equivalent:

```
...
<delete parentXID="1" DataXIDmap="104" position="4"> // !!!
   <child4 />
</delete>
<delete parentXID="1" DataXIDmap="102" position="2">
   <child2 />
</delete>
...
```

In a similar way, we define other *swap* possibilities:

Lemma 4.2.7
$X1$ and $X2$ are *insert* operations with the same parent. If $p1 \geq p2$, then the two operations may be swapped, with $p1$ replaced by $p1 + 1$

The goal is that the position of insert operations should refer to the position of the nodes after the two operations are executed. Consider previous example. We now want to insert the deleted nodes. A possible edit script is:

```
...
<insert parentXID="1" DataXIDmap="104" position="3">
   <child4 />
</delete>
<insert parentXID="1" DataXIDmap="102" position="2">
   <child2 />
</delete>
...
```

It may be transformed into the equivalent:

```
...
<insert parentXID="1" DataXIDmap="102" position="2">
   <child2 />
</delete>
<insert parentXID="1" DataXIDmap="104" position="4"> // !!!
   <child4 />
</delete>
...
```

Lemma 4.2.8
Let S be an edit script, with two consecutive operations: an *insert* $X1$ followed by a *delete* $X2$, with the same parent, and the respective positions $p1$ and $p2$. If $p1 = p2$, then, for each document D such that S is valid with D, S is equivalent to S' obtained by removing the two operations. If $p1 < p2$, then S' is obtained by swapping $X1$ and $X2$, and replacing $p2$ by $p2 - 1$. If $p1 > p2$, then S' is obtained by swapping $X1$ and $X2$, and replacing $p1$ by $p1 - 1$.

Lemma 4.2.9
Let S be an edit script, with two consecutive operations: an *insert* (resp. delete) $X1$, followed by a *delete* $X2$. Suppose that the following condition applies: (i)

$X1$ and $X2$ do not have the same parent, (ii) $X2$ does not delete part of (or an ancestor of) the subtree is inserted by $X1$ (resp. $X2$ do not delete an ancestor of the subtree that is deleted by $X1$). Such two operations are then said *independent operations*. Then S is equivalent to S' obtained by swapping $X1$ and $X2$.

Lemma 4.2.10
Let S be an edit script, with two consecutive operations: an *insert* (resp. delete) $X1$, followed by a *delete* $X2$ (as in previous lemma). If $X2$ deletes part of a subtree inserted by $X1$, then S' obtained by removing $X2$, and modifying $X1$ according to $X2$, is equivalent to S. The modification of $X1$ consists in removing from the inserted data in $X1$ the subtree deleted by $X2$. Note that the XID of corresponding nodes should also be removed from $X1$.

Lemma 4.2.11
Let S be an edit script, with two consecutive operations: an *insert* (resp. delete) $X1$, followed by a *delete* $X2$ (as in previous lemma). If $X2$ deletes an ancestor of a subtree deleted by $X1$, then S' obtained by removing $X1$, and modifying $X2$ accordingly, is equivalent to S. The modification of $X2$ consists in adding in $X2$ the piece of data removed by $X1$. The XID of corresponding nodes should also be added to $X2$.

Conversely, if $X2$ is an *insert* operations, a swap if possible by using similar updates.

A summary is shown in Figure 4.1. Some operations are marked *not used*, meaning that we do not use them in the next section. Intuitively, they correspond to a “correct” order between the two operations, so that we do not swap the two operations.

4.2.3 A Definition for XyDelta

In this section, we first introduce the notion of a XyScript. It is a specific edit script that represents a XyDelta, namely a set of operations. Then we extend the notion of XyDelta, by showing that each edit script is equivalent to some XyScript.

Definition 4.2.12 (XyScript and XyDelta)
A XyScript is an edit script in which atomic operations are sorted according to the COMPARE *function in Figure 4.2. More precisely, operation O_1 occurs before*

X1	X2	Condition	Action to swap
delete	delete	same parent, $p1 \leq p2$	p2<-p2+1
delete	delete	same parent, p1 > p2	not used (p1<-p1-1 if used)
insert	insert	same parent, $p1 \geq p2$	p1<-p1+1
insert	insert	same parent, p1<p2	not used (p2<-p2-1 if used)
delete	insert		not used
insert	delete	same parent, p1=p2	replace with NULL
insert	delete	$X2$ deletes part of the subtree inserted by $X1$	remove $X2$, modify $X1$
insert	delete	$X2$ deletes an ancestor of the subtree inserted by $X1$	remove $X1$, modify $X2$
insert	insert	$X2$ inserts appends a subtree to the one inserted by $X1$	remove $X2$, modify $X1$
delete	delete	$X2$ deletes an ancestor of subtree deleted by $X1$	remove $X1$, modify $X2$
insert	delete	none of the above (independent operations)	swap without modifications
delete	delete	none of the above (independent operations)	swap without modifications
insert	insert	none of the above (independent operations)	swap without modifications

Figure 4.1: Summary of swap operations to reorder Edit Scripts

operation O_2 if and only if

$$COMPARE(O_1, O_2) = true$$

. For each XyScript, that is a sequence *of operations, we name* XyDelta *the* set *of these operations.*

Intuitively, a XyScript is an edit script in which atomic operations are ordered as follows:

1. the *delete* operations come first,
2. *delete* operations with the same parent node are ordered in the reverse order of the deleted nodes positions
3. *insert* operations with the same parent node are ordered in the same order as the inserted nodes positions

The goal of these constraints is that: (i) node's position in *delete* is the same as the position of the node in the initial version of the document, (ii) node's position in *insert* is the same as the position of the node in the final version of the document.

One can show that this is true, i.e. the following theorem is true:

Theorem 4.2.13
The order of operations in a XyScript is such that the position of nodes for each operation corresponds exactly to the initial (for delete) or final (for insert) position of the node in the document.

The proof is not detailed here. The proof is simple since the order (i.e. the `COMPARE` function in Figure 4.2) was defined to match the exact requirements that conduct to initial and final positions of nodes in each operation. This corresponds to the comparisons depicted in the *First Part* of Figure 4.2. This defines a partial order on the set of operations. The second part is used to define a total order on the set of operations.

For instance, consider the deletion of two nodes in a document. A possible edit script is as follows:

```
<delete DataXIDmap="101" parentXID="1" position="1">
   <deleteMe1 />
</delete>
```

```
<delete DataXIDmap="102" parentXID="1" position="1">
   <deleteMe2 />
</delete>
```

After the first node is deleted, the position corresponding to operations on sibling nodes (with higher positions) are decreased by 1. Thus, in the second *delete* operation, the second node position is now 1. This edit script is not a XyScript. A XyScript would be:

```
<delete DataXIDmap="102" parentXID="1" position="2">
   <deleteMe2 />
</delete>
<delete DataXIDmap="101" parentXID="1" position="1">
   <deleteMe1 />
</delete>
```

By reversing the order of delete operations, the position used in each operation correspond to the initial position of the nodes.

Based on the transformations described in previous section, we propose the theorem below:

Theorem 4.2.14 (XyScript existence)
Any script S valid with some document D can be transformed in a XyScript S' equivalent to S with D. Thus, for any script S valid with some document D, there is an equivalent (with D) XyDelta.

This result is obtained by applying the *bubble-sort* algorithm on S. The order relationship used is compliant with the order mentioned previously and is detailed in Figure 4.2. An important aspect is the *Second Part* of Figure 4.2 that was added to obtain a total order on the set of operations. This was necessary to apply the bubble-sort algorithm.

Note that each time a FALSE value is returned, the two operations must be swapped, and are modified accordingly. For the algorithm to be correct, it is necessary to verify that the modifications applied to the operation ensure that the compare function returns TRUE after they have been modified and swapped. This is indeed the case.

The XyScript is an ordered representation of operations. The unordered representation of these operations, namely the set of operations, is the corresponding XyDelta.

At this point, it is also important to note that bubble-sort algorithms have a quadratic time complexity. One can use other techniques that are tailored to specific parts of the applications. For instance, in XyDiff, the sorting is achieved by using the order of nodes in the original and final documents, since they are available when the delta is constructed.

The first part of Figure 4.2 uses information that is contained in the operation descriptors. However, the second part (the extension) uses information (ancestor of nodes) that is not contained in the operation descriptors. It must be obtained from the original document. Thus, to convert an edit script into a XyDelta, the original document is required. In order to manipulate XyDeltas (e.g. aggregation - see below), we chose to also store this information in the header of the delta. More precisely, we store, for each node that is deleted (resp. inserted) the nodes on the path up to the root in the original (resp. final) document, and their position in the original (resp. final) document. To save space, this information is stored using two tree structures in the XyDelta, one for the ancestor's paths of delete operations, and one for the ancestor's paths of insert operations. Each tree summarizes a part of the tree structure of the original (resp. final) document, that is necessary to retrieve the ancestor path descriptors and the position of ancestor nodes.

Remark 4.2.1
[delete+insert] In some cases, there might be an insert operation that inserts some data that has been deleted previously in the edit script. If (part of) the data deleted and then inserted is strictly identical, one may want to remove the corresponding operations. However, it is first necessary to verify that the persistent identifiers (XIDs) are the same on the inserted data than on the deleted data. If not, these two operations are <u>not</u> equivalent to an empty operation.

Remark 4.2.2
[No original document] An important aspect of the algorithm based on Figure 4.2 is that it does not require to use the original document. The proof is obtained by considering all possible tests that are conducted. For most unitary tests that consider the *type* and *position* fields of operations, this is obvious. For some other tests, that consider *ascendant* or *descendant* expressions, there is a precise reason. The reason is that insert and delete operations contain the XID-Map of the subtree below the node on which the operation applied. This information is sufficient to process the tests shown in Figure 4.2. We explain below that for this reason, the

```
boolean COMPARE(Operation X1, Operation X2) {

  // === First Part ===

  // delete operations are before insert operations
  if (X1.Type=="delete")&&(X2.Type=="insert") {
    return TRUE;
  }
  if (X1.Type=="insert")&&(X2.Type=="delete") {
    return FALSE;
  }

  // in the case of two delete operations
  // -descendant subtrees should be deleted first
  // -sibling nodes are deleted first on the right
  if (X1.Type=="delete")&&(X2.Type=="delete") {
    if (X1.node DESCENDANT OF X2.node) return TRUE;
    if (X2.node DESCENDANT OF X1.node) return FALSE;
  }

  // in the case of two insert operations
  // -ancestor nodes should be inserted first
  // -sibling nodes are inserted first on the left
  if (X1.Type=="insert")&&(X2.Type=="insert") {
    if (X2.parent DESCENDANT OF X1.node)
       OR (X2.parent EQUALS X1.node) return TRUE;
    if (X1.parent DESCENDANT OF X2.node)
       OR (X1.parent EQUALS X2.node) return FALSE;
  }

  // === Second Part ===
  // This part is necessary for a total order

  Let P be the first common ancestor of X1 and X2
  Let P be the first common ancestor of X1.node and X2.node
  // 'first' mean closest to X1.node and X2.node
  // for instance, P may be the common par-
ent of X1.node and X2.node

  Let PX1 be the ancestor of X1.node (or X1.node) that is a child of P
  Let PX2 be the ancestor of X2.node (or X2.node) that is a child of P

  if (X1.type="delete")&&(X2.type="delete") {
      return (PX1.Position>PX2.Position);
  }
  else if (X1.type="insert")&&(X2.type="insert") {
      return (PX1.Position<PX2.Position);
  }

}
```

Figure 4.2: Compare function to find the XyDelta order of Edit Scripts

move operation must contain the XID-Map of the moved subtree, although it does not seem necessary at first glance.

Corollary 4.2.15
Let S_1 and S_2 be two XyScripts, such that S_1 is valid on D, $S_1(D) = D'$ and S_2 is valid on D'. There is a XyScript S that is valid on D and equivalent to the aggregation of S_1 and S_2.

It is possible to create a XyDelta aggregation algorithm that does not require to use the original document. It can be done as follows:

1. Output as a sequence of operations the Edit Scripts corresponding to XyDelta S_1 and S_2
2. Merge the two sequences of operations as an Edit Script E
3. Find the XyScript for E

Summary We proved that any edit script is equivalent to a XyScript, and thus may be represented by a XyDelta. A XyDelta is a *set* representation of changes between two documents. Given a XyDelta, one can construct the corresponding XyScript. XyScripts are edit scripts that have specific properties respectively to the attribute of their operations.

4.3 Extending the Model

In this section, we propose several extensions to the XyDelta model. Some of them are used to improve the support of XML data (e.g. text nodes, attributes), others enrich the update mode (e.g. the move operation).

4.3.1 Text nodes and Attribute nodes

To add the support of text nodes, we simply consider text nodes as nodes with no children. The support of attributes is done in the spirit of XML:

1. attribute nodes are attached to some element node
2. attribute nodes have no XID: they are identified by the pair consisting of: (i) the XID of their element node and (ii) their attribute name

The attribute operations consist in adding an attribute (name and value) to some element, removing an attribute name and value, or modifying the value of some attribute. They are described as follows:

```
<attr-insert
    nodeXID   = "..."
    attrName  = "..."
    attrValue = "..." />
<attr-delete
    nodeXID   = "..."
    attrName  = "..."
    attrValue = "..." />
<attr-modify
    nodeXID   = "..."
    attrName  = "..."
    OldValue  = "..."
    NewValue  = "..." />
```

As in previous section, note that the insert and delete operations are exactly symmetrical. In particular, the value of the attribute is stored along with the delete operations, although this was not strictly necessary, in the context of edit scripts that go only forward.

Note also that the order of attribute operations is not very important. However, in the context of edit scripts, it is important that attribute operations only occur when their element node exists, i.e. before it has been deleted or after it has been inserted (if any of these two operations occurs).

Others . We do not detail here how to handle specific XML concepts such as entities, comments, CDATA vs. PCDATA. As previously mentioned, the proper handling of white-spaces within text representations of XML may become a difficult task due to variations between approaches such as DOM, SAX, XQuery, XPath, ...

4.3.2 Adding Update and Move operations

In this section, we present *update* and *move* extensions to the model. But first, we introduce the notion of *matching* between the nodes of the original and final documents.

Consider a node that is present in the two versions of the document, with the same XID. It has not been deleted, it has not been inserted. We say that the two versions of that node define a *matching*. A node is matched if we can find his corresponding node in the other version of the document.

Update *Update* operations represent the changing value of a text node that is attached to some element node. We consider update operations if and only if: (i) an element node a is matched between the two versions of the document and (ii) it has a single child node that is a text node. In this case, if the text node value changes, there is exactly one insert and one delete operation with the parent XID corresponding to a. We annotate these two operations with an attribute `update="yes"` indicating that the pair corresponds in fact to a single *update* operation. Based on this, it is possible to improve furthermore the syntax of update by writing, in the XML document representing the delta, a single `update` operation for each pair. Only when the delta is processed, the update operation may be split into the corresponding insert and delete pair (if necessary to execute it). The syntax is as follows:

```
<update
    parentXID="..."
    OldDataXIDmap="(...)"
    NewDataXIDmap="(...)" >

    <OldText> This is the old text node value </OldText>
    <NewText> This is the new text node value </NewText>

</update>
```

Note that the position is not listed since it is always 1 by definition. Indeed, remind that we only consider the update of text as a single child node. We do not consider, for instance, the update of some text in:

```
<root>
```

```
  <x/>
  text
  <y/>
</root>
```

In some applications, the system may keep the same XID for the changing text node. In other words, the semantics is that the text node is the same although its value changes. In that case, the `OldDataXIDmap` and *NewDataXIDmap* attributes have the same value. They may be grouped within a single `DataXIDmap` attribute. In other applications, the XIDs are used to quickly reconstruct versions of the documents, or to index the textual content of document. Thus, the system may prefer to assign a new XID to text nodes each time their value changes.

In some XML documents, the text nodes may contain very long strings. In that case, a possible extension is to use string diffs (or text-based diffs) on them. This results in shorter deltas, and describes changes with slightly more accuracy from a semantic point of view.

Move As we have seen in previous chapter, *move* operations are important both for the efficiency and the semantics of the deltas. Our model has a move operation. This distinguishes from other work without move. It turns out that for some purposes, it is useful to cut it in delete/insert but semantically it is a move. The semantic of a move is different than a pair of delete and insert in that it keeps the persistent identifiers of nodes.

We handle move operations in a way that is similar to updates. More precisely, a pair of insert and delete operations that removes and then inserts the same data, with the same XIDs, is annotated as a move operations using a `move="yes"` attribute. Again, the storage of move operations may be achieved using a single XML subtree as follows:

```
<move
    sourceParentXID="..."
    sourcePosition ="..."
    targetParentXID="..."
    targetPosition ="..."
    DataXIDmap     ="(...)" >

</move>
```

Two important things may be noted:

- The `DataXIDmap` attribute may seem unnecessary. Indeed, storing the XID of the subtree root node is in some cases sufficient. However, this `DataXIDmap` is necessary, as seen previously, to compare and order the operations of the corresponding edit script, for instance to aggregate with some other XyDelta.

- It is in general not necessary to store the data that is moved. Indeed, the data is present both in the source and target version of the document. Thus, given one of the two versions, it is always possible to reconstruct the other version.

It is important here to mention that it is the role of the creator of the delta to decide whether a pair of delete and insert is a move or not. More precisely, consider some fragment of XML tree that is present in both documents. Depending on various criteria, the change detection tool may decide: (i) that this data is persistent from one version to another and has been moved (if parents are different), then the data keeps the same XIDs (ii) that the two data sets are different data fragments, although having an identical value. In other words:

(i) the identity between data fragments is defined based on the XIDs, and not on text value and element names,

(ii) the semantic of move is to consider the moving some data around the document, keeping the same XIDs

(iii) it is the change detection algorithm who decides whether data fragments with identical values (text nodes, tag names) have the same XIDs

Cutting *Move* A move operations consists both in deleting a part of the document and inserting it in another place. Since our model uses precises rules with respect to the nodes positions and order of operations, it turns out that the proper processing of move is achieved but cutting it into a “special” pair of delete/insert. For instance, when aggregating deltas, the `to` position of moves from the first delta, and the `from` position of moves from the second delta have to be updates, which is easily achieved by considering the corresponding insert and delete operations. Semantically, the “special” pair of delete/insert is a move, in particular it keeps the persistent identifiers of nodes.

4.4 Properties of XyDelta

In this section, we present various properties of XyDeltas, and mention briefly the corresponding algorithms and their cost.

XyDelta: Summary Intuitively, a XyDelta is an XML file that represent a set of operations transforming one XML document into another. A XyScript is an edit script. A XyDelta represents a set of operations which, when ordered correctly, provides a XyScript. The operations descriptors, and in particular the nodes identifiers and positions, refer to the first version of the document (for delete) or to the second version (for insert). Update and move operations are supported.

Creating a XyDelta There are several ways to create a XyDelta.

- One is to construct a set of operations given the two versions of the document. This is done typically by *XML diff* programs. The algorithms and cost may vary. A comparative study is conducted is Chapter 5.

- A second possibility is to start from a document and generate the set of operations. The operations may be generated randomly, for instance in a changes simulator that we implemented, or they may be generated based on users actions, for instance in an XML editor.

- A third possibility is, given an edit script, to transform it in the corresponding XyDelta. We proved that this is always possible using a bubble-sort algorithm, which cost is quadratic. The comparison of two operations has a $O(l)$ cost, where l is the maximum length of their XID-Map. Thus, the sorting cost would be $O(n^2 * l)$. However, one can use faster sorting algorithms, such as quick-sort in $O(n * log(n))$. The modifications of nodes positions are then computed on position tables that are stored for each node. The cost is in $O(l * n * (log(n) + log(f)))$ where f is the maximum number of nodes with the same parent.

Inverse of a XyDelta Thanks to the exact symmetry between insert and delete operations, the inverse of a XyDelta always exists. It is obtained by simply renaming each *insert* into delete, and each *delete* into insert. For move (resp. update), it suffices to think of move (resp. update) as a pair of delete and insert to obtain the inverse. The space cost is a constant, and the time cost is linear in the number of operations.

Applying a XyDelta to a document To apply a XyDelta to a document, we first generate the corresponding edit script by ordering the set of operations according to the rules presented previously. Note that positions are *not* modified since, by definition of XyDelta, they already correspond to the correct positions according to this order. In particular, delete operations are executed first, and then insert operations. Obviously, we keep the subtrees which are deleted as part of a *move* operation until they are inserted again. We can then use it when executing the *insert* part of the *move*. The cost of ordering a XyDelta is in $O(l * n * log(n))$, where n is the number of operations, and l the maximum length of an XID-Map in the delta.

Aggregating XyDeltas To aggregate XyDelta, we first generate the two corresponding edit scripts, and then we find the XyDelta corresponding to their concatenation. The cost is the sum of the cost of ordering the operations in the two deltas, plus the cost of finding the XyDelta. Thus, the total cost is $O(l*(n_1+n_2)*log(f))$, where $n1$ and $n2$ are the number of operations in each delta, l the maximum length of an XID-Map in these deltas, and f the maximum number of nodes with the same parent.

4.5 Conclusion

We have presented a formalism for representing XML changes in XML. The XyDelta that we obtain have nice mathematical properties.

This framework has been implemented as part of the XyDiff [34] project.

We believe that further work is necessary on the topic of manipulating several deltas to manage the history of a document, retrieve versions and query changes. Marian worked on the topic of storage strategies based on deltas. Other approaches would be interesting, such as storing an extended version of the document with time information [23], and generating deltas on the fly when necessary.

Chapter 5

A comparative study of XML diff tools

Abstract. *Change detection is an important part of version management for databases and document archives. The success of XML has recently renewed interest in change detection on trees and semi-structured data, and various algorithms have been proposed. We study here different algorithms and representations of changes based on their formal definition and on experiments conducted over XML data from the Web. Our goal is to provide an evaluation of the quality of the results, the performance of the tools and, based on this, provide guidelines to users for choosing the appropriate solution for their applications.*

I started this work and lead it in a cooperation with Talel Abdessalem and Yassine Hinnach.

5.1 Introduction

The context for the present work is change control in XML data warehouses. In such a warehouse, documents are collected periodically, for instance by crawling the Web. When a new version of an existing document arrives, we want to understand changes that occurred since the previous version. Considering that we have only the old and the new version for a document, and no other information on what happened between, a *diff* needs to be computed. A typical setting for the *diff* algorithm is as follows: the input consists in two files representing two versions of the same document, the output is a *delta* file representing the changes that occurred between them.

In this chapter, we analyze different proposals. We study two dimensions of the problem: (i) the representation of changes (ii) the detection of changes.

(i) Representing Changes To understand the important aspects of changes representation that we consider in this chapter, we recall briefly some possible applications:

- In Version management [29, 69], the representation should allow for effective *storage strategies* and efficient *reconstruction of versions* of the documents.
- In Temporal Applications [26], the support for a *persistent identification* of XML tree nodes is mandatory since one would like to identify (i.e. trace) a node through time.
- In Monitoring Applications [27, 84], changes are used to detect events and trigger actions. The trigger mechanism involves *queries on changes* that need to be executed in real-time. For instance, in a catalog, finding the product of type 'digital camera' and of which the price has decreased.

The *deltas* we consider here are XML documents summarizing the changes. The choice of XML is motivated by the need to exchange, store and query these changes. Since XML is a flexible format, there are different possible ways of representing the changes on XML and semi-structured data [23, 63, 69, 114], and build version management architectures [29]. In Section 5.3, we compare several change representation models.

The results of our study indicate two main directions to represent changes:

(i) one is to summarize the two versions of the documents and add change information (e.g. *DeltaXML* [63], *XDL* [76]). The change information represents the operations that transform one version into another.

(ii) the other is to focus on edit operations, i.e. to give a list of edit operations that transform one version into another (e.g. *XUpdate* [114], *XyDelta* [69])

The advantage of (i) is that the summary of the documents gives a useful context to understand change operations. The drawback of (i) is that it (often) lacks a formal model or mathematical properties (e.g. aggregation), in particular, no precise framework for version management or even querying has been developed.

On the other hand, (ii) may provide a better formal model of edit operations. In (ii), the delta uses identifiers to refer to nodes of the document. The drawback of (ii) is that such deltas do not contain sufficient information for monitoring or to support queries , i.e. the document has to be processed. Thus, they are less intuitive to read and use in simple applications.

(ii) Change Detection In some applications (e.g. an XML document editor) the system knows exactly which changes have been made to a document, but in our context, the sequence of changes is unknown. Thus, the most critical component of change control is the *diff* module that detects changes between an old version of a document and the new version. The input of a *diff* program consists in these two documents, and possibly their DTD or XMLSchema. Its output is a *delta* document representing the changes between the two input documents. Important aspects are as follow:

- *Correctness:* A diff is *correct* if it finds a set of operations that is sufficient to transform the old version into the new version of the XML document. In other words, a diff is correct if it misses no changes. For some application, one may want to trade correctness for performance, for compactness of the delta or to focus only on certain changes. This is not considered here, all diff algorithms we present here are correct.

- *Minimality:* In some applications, the focus will be on the minimality of the result (e.g. number of operations, edit cost, file size) generated by the *diff*. This notion is explained in Section 5.2. Minimality of the result is important to save storage space and network bandwidth. Also, the effectiveness of version management depends both on minimality and on the representation of changes.

- *Semantics:* Some algorithms consider more than the tree structure of XML documents. For instance, they may consider keys (e.g. ID attributes defined in the DTD) and match with priority two elements with the same tag if they have the same key. For instance, a `product` node in a `catalog` is identified by the value of its `name` descendant node. In the world of XML, the semantics of data is becoming extremely important [103] and some applications may be looking for semantically correct results or impose semantic constraints. For instance it is considered correct to update the `price` node of a product, but it may be considered incorrect to update the product's `reference` node (product identifier).

- *Performance and Complexity:* With dynamic services and/or large amounts of data, good performance and low memory usage become mandatory. For example, some algorithms find a minimum edit script (given a cost model detailed in Section 5.2) in quadratic time and space, whereas others run in linear time (and space).

- *"Move" Operations:* The capability to detect *move* operations (see Section 5.2) is only present in certain *diff* algorithms. The reason is that it has an impact on the complexity (and performance) of the *diff* and also on the minimality and the semantics of the result.

To explain how the different criteria affect the choice of a *diff* program, consider some cooperative work application on large XML documents. The large XML documents are replicated over the network. We want to enable concurrent work on these documents and efficiently update the modified parts. Thus, a *diff* between XML documents is computed and the delta is used to broadcast changes to the various copies. Then, changes can be applied (propagated) to the files replicated over the network. When a "small" diff is propagated, (i) the bandwidth cost is recuded, and (ii) it is less likely that it conflicts with an update on a different site. On the other hand, if there is little replication (and little risk of conflicts), it is more important to compute the diff fast than to minimize its size. Note that with some diff programs, it is even possible to use the support of keys (e.g. support of DTD ID attributes) to divide the document into finer grain structures, in order to implement precise locking mecanisms and handle more efficiently concurrent transactions.

Our study considers several possible design strategies for XML diff tools. We will see the cost of quadratic algorithm, e.g. *MMDiff* doesn't scale up to megabyte files. We also show the impact of the use of greedy rules (e.g. *XyDiff*) that save time, but decrease the minimality of results. Using simple examples, we detail how the combination of quadratic algorithm with preprocessing steps (e.g. pruning the tree) generates high quality (but not minimal) results. We will also mention important features of diff tools that allow to improve the accuracy of results.

Experiment Settings Our comparative study relies on experiments conducted over XML documents found on the web. Xyleme crawler [119] was used to crawl more than five hundred millions web pages (HTML and XML) in order to find five hundred thousand XML documents. Because only part of them changed during the time of the experiment (several months), our measures are based roughly on hundred thousand XML documents. Only some experiments were run on the entire set. Most were run on sixty thousand documents because of the time it would take to run them on all the available data. It is also interesting to run it on private data (e.g. financial data, press data). Such data is typically more regular. For instance, we ran our tests on several versions of XML data from DBLP [66]. We intend to conduct other experiments in the future.

Remark Our work was done primarily for XML documents. It can also be used for HTML documents by first XML-izing them. For instance, a relatively easy task consists in properly closing tags. However, change management (detection+representation) for a "true" XML document [1] is semantically much more informative than for HTML. It includes pieces of information such as the insertion of particular subtrees with a precise semantics, e.g. a new product in a catalog.

The rest of the chapter is organized as follows. First, we present the data model, operation model and cost model in Section 5.2. Then, we compare change representations in Section 5.3. In Section (5.4), we compare change detection algorithms and their implementation programs. In Section 5.5, we present a performance analysis (time and space). Finally, we study the quality of the results of diff programs in Section 5.6. The last section concludes the chapter.

5.2 Preliminaries

In this section, we introduce the notions that will be used along the chapter. The data model we use for XML documents is labeled ordered trees as in [69]. We also mention some algorithms that support unordered trees.

Operations The change model is based on editing operations as in [69], namely *insert, delete, update* and *move*. There are two possible interpretations for these operations: Kuo-Chung Tai's model [99] and Selkow's model [97].

[1] or for an HTML document that has been XML-ized using advanced techniques

In [99], deleting a node means making its children become children of the node's parent. For instance, deleting `<product>` in the subtree

```
<catalog><product><price value="\$99"/></product></catalog>
```

yields the result

```
<catalog><price value="\$99"/></catalog>
```

This model may not be appropriate for XML documents, since deleting a node changes its depth in the tree and may also invalidate the document structure according to its DTD (or XMLSchema). In general, this model is not appropriate for object models [2] where the type of objects and relations between them is important. It seems more appropriate, for instance, to applications such as biology where XML is used to represent DNA sequences [107].

Thus, for XML data, we use Selkow's model [97] in which operations are only applied to leaves or subtrees. In particular, when a node is deleted, the entire subtree rooted at the node is deleted. This captures the XML semantic better, for instance removing a product from a catalog by deleting the corresponding subtree. Important aspects presented in [69] include (i) management of positions in XML documents (e.g. the position of sibling nodes changes when some are deleted), and (ii) consistency of the sequence of operations depending on their order (e.g. a node can not be updated after one of its ancestors has been deleted).

Edit Cost The *edit cost* of a sequence of edit operations is defined by assigning a cost to each operation. Usually, this cost is 1 per node touched (inserted, deleted, updated or moved). If a subtree with n nodes is deleted (or inserted), for instance using a single delete operation applied to the subtree root, then the edit cost for this operation is n. Since most *diff* algorithms are based on this cost model, we use it in this study. The *edit distance* between document A and document B is defined by the minimal edit cost over all edit sequences transforming A in B. A *delta* is *minimal* if its edit cost is no more than the edit distance between the two documents.

One may want to consider different cost models. However, some cost models imply a trivial solution of the diff problem. Consider for instance the case of assigning $cost = 1$ for each edit operation, e.g. deleting or inserting an entire subtree. When two documents are different, a minimal edit script would often consist

[2]see Document Object Model (DOM) for XML, http://www.w3.org/DOM/

in the following pair of operations: (i) delete the first document with a single *delete* operation applied to the document's root (ii) insert the second document with a single *insert* operation.

We briefly mention in Section 5.6 some results based on a cost model where the cost for *insert* , *delete* and *update* is 1 per node, whereas the cost for *moving* an entire subtree is only 1 (see next).

The *move* operation The semantics of *move* is to identify nodes (or subtrees) even when their context (e.g. ancestor nodes) has changed. Some of the proposed algorithms are able to detect *move* operations between two documents, whereas others do not. We recall that most formulations of the change detection problem with *move* operations are NP-hard [126]. So the drawback of detecting *moves* is that the algorithms that can be used in practical time will only approximate the minimum edit script. In [126], they consider the problem of comparing two CUAL (Connected, Undirected, Acyclic and Labeled) graphs. By reduction from exact cover by 3-sets, one can show that finding the distance between two graphs is NP-hard. They extend this by proposing a constrained distance metric, called the degree-2-distance, requiring that any node to be inserted (deleted) has no more than 2 neighbor. In this view, Selkow's model corresponds to finding the degree-1 distance.

The improvement when using a *move* operation is that, in some applications, users will consider that a *move* operation is more intuitive (or less costly) than a *delete* and *insert* of the subtree. It often corresponds to reality, e.g. in a storage file-system, moving a directory of files is cheaper than copying (and then deleting) them, and also cheaper than moving each file one-by-one.

In temporal databases, *move* operations are important to detect from a semantic viewpoint because they allow to identify (i.e. trace) nodes through time better than *delete* and *insert* operations.

Mapping/Matching In the next sections, we will also use the notion of "mapping" between the two trees. Each node in A (or B) that is not deleted (or inserted) is "matched" to the corresponding node in B (or A). A *mapping* between two documents represents all matchings between nodes from the first and second documents. In some cases, a *delta* is said “minimal” if its edit cost is minimal for the restriction of editing sequences compatible with a given “mapping”[3].

The definition of the mapping and the creation of a corresponding edit sequence are part of the “change detection”. The “change representation” consists in a data model for representing the edit sequence.

5.3 Change Representation models

XML has been widely adopted both in academia and in industry to store and exchange data. [26] underlines the necessity for querying semistructured temporal data. Recent works [26, 63, 29, 69] study version management and temporal queries over XML documents. Although an important aspect of version management is the representation of changes, a standard is still missing.

In this section we recall the issues in change representation for XML documents, and we present the main recent proposals on the topic, namely *DeltaXML* [63], *XDL* [76], *XUpdate* [114] and *XyDelta* [69]. Then we give a summary of the different formats, their features and equivalences between them. Finally, we present some experiments conducted over Web data.

As previously mentioned, the main motivations for representing changes are: version management, temporal databases and monitoring data. Here, we analyze these applications in terms of (i) versions storage strategies and (ii) querying changes.

Versions Storage Strategies In [28], a comparative study of version management schemes for XML documents is conducted. For instance, two simple strategies are as follow : (i) storing only the latest version of the document and all the deltas for previous versions (ii) storing all versions of the documents, and computing deltas only when necessary. When only deltas are stored, their size (and edit cost) must be reduced. For instance, the delta is in some cases larger than the versioned document. We have analyzed the performance for reconstructing a document’s version based on the delta. The time complexity is in all cases linear in the edit cost of the delta. The computation cost for such programs is close to the cost of manipulating the XML structure (reading, parsing and writing).

[3]A sequence based on another mapping between nodes may have a lower edit cost

One may want to consider a flat text representation of changes that can be obtained for instance with the Unix diff tools. In most applications, it is efficient in terms of storage space and performance to reconstruct the documents. Its drawback are: (i) that it is not XML and can not be used for queries (ii) files must be serialized into flat text and this can not be used in native (or relational) XML repositories.

Querying Changes We recall here that support for both indexing and persistent identification is useful. On one hand, labeling nodes with both their prefix and postfix position in the tree allows to quickly compute ancestor/descendant tests and thus significantly improves querying [8]. On the other hand, labeling nodes with a persistent identifier accelerates temporal queries and reduces the cost of updating an index. In principle, it would be nice to have one labeling scheme that contains both structure and persistence information. However, [36] shows that this requires longer labels and uses more space.

Also note that using *move* operations is often important to maintain persistent identifiers since using *delete* and *insert* does not lead to a persistent identification. Thus, the support of *move* operations improves the effectiveness of temporal queries.

5.3.1 Change Representation models

We now present change representation models. On one hand, the XML delta formats *XUpdate* [114] and *XyDelta* [69] describe the list of operations that transform one document into another. In these deltas, the nodes of the source and target documents are identified using XPath expressions (*XUpdate*) or XIDs (*XyDelta*). On the other hand, the XML delta format *DeltaXML* [63] gives a summary of the source document. This summary is enriched by adding specific elements and attributes to describe the operations that transform it into the target document (details below). *XDL* [76] can be used in these two different ways.

Our examples of *deltas* represent the changes that occurred between the two versions of the document presented in Figures 5.1 and 5.2 (pages 84, 85). The notion of XID and XDL Path is detailed below. Note also that the problem of ignorable white spaces is a technical issue that sometimes becomes hard to deal with. In this section, we ignore this issue (without loss of generality) and we only consider the “real” content of XML documents.

```
                          | XID | XDL Path
<catalog>                 |  15 | .
  <product>               |   7 | /1
    <name>                |   2 | /1/1
      Notebook            |   1 | /1/1/1
    </name>               |     |
    <description>         |   4 | /1/2
      2200MHz Pentium4    |   3 | /1/2/1
    </description>        |     |
    <price>               |   6 | /1/3
      $1999               |   5 | /1/3/1
    </price>              |     |
  </product>              |     |
  <product>               |  14 | /2
    <name>                |   9 | /2/1
      Digital Camera      |   8 | /2/1/1
    </name>               |     |
    <description>         |  11 | /2/2
      Fuji FinePix 2600Z  |  10 | /2/2/1
    </description>        |     |
    <status>              |  13 | /2/3
      Not Available       |  12 | /2/3/1
    </status>             |     |
  </product>              |     |
</catalog>                |     |

(Note: XID and XDL-Path are node identifiers)
```

Figure 5.1: First version of a document

DeltaXML In [63] (or similarly in [26]), the delta information is stored in a "summary" of the original document by adding "change" attributes. It is easy to present and query changes on a single delta, but slightly more difficult to aggregate deltas or issue temporal queries on several deltas. The delta has the same look and feel as the original document, but it is not strictly validated by the document's DTD. The reason is that while most operations are described using attributes (with a `deltaxml` namespace), a new type of tag is introduced to describe text nodes updates. More precisely, for obvious serialization/parsing reasons, the old and new values of a text node cannot be put side by side, and the tags `<deltaxml:oldtext>` and `<deltaxml:newtext>` are used to distinguish them. A specific DTD is generated for each input document DTD.

```
<catalog>
  <product>
    <name>Notebook</name>
    <description>2200MHz Pentium4</description>
    <price>$1999</price>
  </product>
  <product>
    <name>Digital Camera</name>
    <description>Fuji FinePix 2600Z</description>
    <price>$299</price>
  </product>
</catalog>
```

Figure 5.2: Second version of the document

```
<catalog
    xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-
delta-v1"
    deltaxml:delta="WFmodify" >
    <product deltaxml:delta="unchanged"/>
    <product deltaxml:delta="WFmodify">
        <name deltaxml:delta="unchanged"/>
        <description deltaxml:delta="unchanged"/>
        <deltaxml:exchange>
            <deltaxml:old>
                <status deltaxml:delta="delete">Not Avail-
able</status>
            </deltaxml:old>
            <deltaxml:new>
                <price deltaxml:delta="add">$299</price>
            </deltaxml:new>
        </deltaxml:exchange>
    </product>
</catalog>
```

Figure 5.3: DeltaXML delta

There is some storage overhead when the change rate is low because: (i) position management is achieved by storing the root of unchanged subtrees (ii) change status is propagated to ancestor nodes. The delta correspondig to Figures 5.1 and 5.2 is presented in Figure 5.3.

Note that it is also possible to store the whole document, including unchanged parts, along with changed data.

XyDelta In [69], every node in the original XML document is given a unique identifier, namely XID (see Figures 5.1 and 5.2), according to some identification technique called *XidMap*. The *XidMap* gives the list of all persistent identifiers in the XML document in the postfix order of nodes (i.e. descendant first). Then, the delta represents the corresponding operations: identifiers that are not found in the new (old) version of the document correspond to nodes that have been deleted (inserted)[4]. The example in Figures 5.1 and 5.2 is as follows: nodes 12-13 (i.e. from 12 to 13) that have been deleted are removed from the *XidMap* of the second version, while new identifiers (e.g. 16-17) are assigned to inserted nodes. The delta corresponding to Figures 5.1 and 5.2 is:

```
<xydelta
  v1_XidMap="(1-15)"
  v2_XidMap="(1-11;16-17;14-15)">
  <delete xid="(12-13)" parent="14" position="3">
    <status>Not Available</status>
  </delete>
  <insert xid="(16-17)" parent="14" position="3">
    <price>$299</price>
  </insert>
</xydelta>
```

As shown in Chapter 4, *XyDeltas* have nice mathematical properties, e.g. they can be aggregated, inverted and stored without knowledge about the original document. Also the persistent identifiers and *move* operations are useful in temporal applications. The drawback is that a *XyDelta* does not contains contexts (e.g. the content and value of ancestor nodes or siblings of nodes that changed) which are sometimes necessary to understand the meaning of changes or present query results to the users. Therefore, the context has to be obtained by processing the document.

XUpdate [114] provides means to update XML data, but it misses a more precise framework for version management or to query changes. In the same spirit as *XyDelta*, it describes the edit operations. The difference is that nodes are identifyied by a path expression instead of their postfix position. While readability is

[4]Move and update operations are described in Chapter 4 (see [69]).

improved, the original document is still necessary to know the exact context of operations. Since path expression can be very long, the size of the delta may be larger than other formats. On the other hand, these path expressions allow to efficiently monitor changes in specific parts of the document. Note also that it allows the use of variables, but a more precise proposal is clearly needed for such an advanced feature. Another drawback is that *XUpdate* does not support backward deltas. The delta correspondig to Figures 5.1 and 5.2 is:

```
<xupdate:modifications version="1.0"
 xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select="/catalog[1]/product[2]/description[1]" >
    <xupdate:element name="price">
      $299
    </xupdate:element>
  </xupdate:insert-after>
  <xupdate:remove select="/catalog[1]/product[2]/status[1]" />
</xupdate:modifications>
```

Microsoft XDL XML Diff Language (XDL) [76] is a proprietary XML-based language for describing differences between two XML documents. An instance of XDL is called the *XDL diffgram*. This language is the most recent proposal. In the spirit of *XUpdate* and *XyDelta* , *XDL* only describes the edit operations. However, the "target" position of insert operations relies on the diffgram tree structure (in the spirit of *DeltaXML*). More precisely, the position where a node is inserted is given by the position of the corresponding operation in the diffgram[5].

On the other hand, the "source" nodes (e.g. deleted nodes) are identified using path descriptors. The path descriptor language is not XPath: the element nodes name is not mentioned, only the node's positions are listed. For instance, `./2` refers to the second child of a node. This results on average in shorter path descriptors than XPath. However, the drawback compared to *XUpdate* is that the path descriptors are not sufficient to monitor changes (the document has to be processed). A *node match* command is also used to identify a node and declare it the "root" of a local context. For instance, the path descriptor:

```
<xd:delete match="/2/3" />
```

[5]In XUpdate or XyDelta, a node identifier is used to describe the target position

may be replaced by

```
<xd:node match="2">
  <xd:delete match="3" />
</xd:node>
```

Note that in this case, the identification of “source” nodes is similar to *DeltaXML*, since the diffgram tree structure summarizes the tree structure of the original document.

As in *XyDelta*, *move* operations are supported. They are described using an additional operation descriptor that connect a pair of *insert* and *delete* operations. A *copy* operation is also used to save space in some cases. XDL is validated by a DTD. Again, a drawback is that backward deltas are not supported. Like *DeltaXML*, the model lacks more precise version management features (aggregation, persistent identifiers). A nice feature of XDL is the use of a signature (hash value) to identify the source document on which the diffgram can be applied. This feature could easily be added to the other formats. The diffgram (delta) corresponding to Figures 5.1 and 5.2 is:

```
<xd:xmldiff
  srcDocHash="......"
  xmlns:xd="http://schemas.microsoft.com/xmltools/2002/xmldiff" >
  <xd:node match="1">
    <xd:node match="2">
      <xd:change match="3" name="price">
        <xd:change match="1">$299
        </xd:change>
      </xd:change>
    </xd:node>
  </xd:node>
</xd:xmldiff>
```

An important difference with other change representation language is that the insert and delete operations are defined according to Tai’s model (see Section 5.2). The set of operations is larger than for other languages which use Selkow’s model, where deleted and insert operations are only applied on leaves or subtrees. However, other languages can represent Tai’s operations as a composition of their own operations. For instance, Tai's delete of a node n is replaced by deleting the entire subtree rooted at n, and their inserting the child subtrees of n as children of the parent of n. A drawback, in that case, is that the persistent identification of nodes is lost. With languages that support move operations, e.g. XyDelta, the persistent identification of nodes is maintained.

Others Languages *Dommitt* [46] representation of changes is in the spirit of *DeltaXML*. However, surprisingly, instead of using change attributes, new node types are created. For instance, when a `book` node is deleted, a `xmlDiffDeletebook` node is used. A drawback is that the delta (and its DTD) is significantly different from the original documents (and their DTD).

5.3.2 Summary

During this study, we tested a few other change formats that are not mentioned here. Some are comparable to those presented here, others turned out to be too limited to really support the tree structure of XML. In the context of applications that we mentioned previously, we summarize next the important aspects of the change formats.

- Monitoring: *DeltaXML* and *XUpdate* make it easier to monitor changes because the deltas contain the name and value of all ancestor nodes of nodes that changed. Thus, processing of simple path queries on changing nodes can be easily supported. Only *DeltaXML* proposes a method for having both the changed and unchanged data in a single XML document. This is important for any uses where the unchanged data is needed also, for instance displaying changes to a user in the context of unchanged data.

- Temporal Queries: Only *XyDelta* manages explicitely the persistant identifiers. For other formats, the identifiers have to be infered by processing the deltas and the source documents, which is costly. We believe that such identifiers should be used to improve the efficiency of temporal queries. An identification mecanism could be stored along with the documents as it is done currently in *XyDelta*, where an *XidMap* file is created for each document.

- Storage: Experiments (see Section 5.3.3) show that *XUpdate*, *XyDelta* and *XDL* save storage space when few changes occur on the document.

> The reason is that these formats only list change operations, whereas the *DeltaXML* gives a summary of the full document. However, the storage space is similar when a fair amount of changes occurs (e.g. 15 percent).

In all cases, validation by a DTD (or XMLSchema) is proposed or may be obtained. It would be interesting to have deltas that are validated by the exact same DTD as the document, but as mentioned previously, a difficulty is then to describe the updates of text nodes (e.g. DeltaXML).

Another important aspect to consider is the formal equivalence (or not) between the various change representations. We focus here on the core definitions of the XML formats presented previously. For instance, we ignore advanced features of *XUpdate* (e.g. the use of variables) since they are not clearly defined (and we found no implementation or tools that use them). In this case, *DeltaXML* and *XUpdate* are equivalent. *XDL* subsumes them. It adds information about “move” operations defined as pair of insert and delete operations. *XyDelta* subsumes *XDL*. It adds information to *delete* operations that can be used to apply the delta backward.

However, all formats could be easily extended to support “move” operations as well as “backward” deltas. In a similar way, all formats could be easily extended to support nice features such as *XDL* verification of the source document using a hash value. Thus, one may want to consider all these formats as “almost” equivalent. So, we believe that any one of them could serve as a basis of a standard for representing XML changes.

5.3.3 Change Representation Experiments

In this section, we present experiments on the space usage of change representations. Consider a given set of operations transforming one document into another. Our experiments conducted over a few thousand files showed that the size of the various XML formats is essentially similar, up to a constant factor. This factor (roughly 2) should not be considered important since it depends on the XML storage architecture (serialized files, native XML or relational).

We noted that *XDL* deltas obtained using Microsoft Diff and Merge Tool [77] are sometimes a bit smaller than others. In most cases, the reason is that the change model is different, in particular it allows *insert* and *delete* operations according to Tai’s model (see Section 5.2).

To illustrate our work, we compare in Figure 5.4 (page 91) the space usage for the two main approches: list of operations vs. summary of the documents. Figure 5.4 shows the size of a *delta* represented using *DeltaXML* or *XyDelta* as function of the edit cost of the delta. The delta cost is defined according to the "1 per node" cost model presented in Section 5.2. Each dot represents the average[6] delta file size for deltas with a given edit cost. It confirms clearly that *DeltaXML* is larger for lower edit costs because it describes many unchanged elements. On the other hand, when the edit cost becomes larger, its size is comparable to *XyDelta*. The *deltas* in this figure are the results of more than twenty thousand XML diffs, roughly twenty percent of the changing XML that we found on the web.

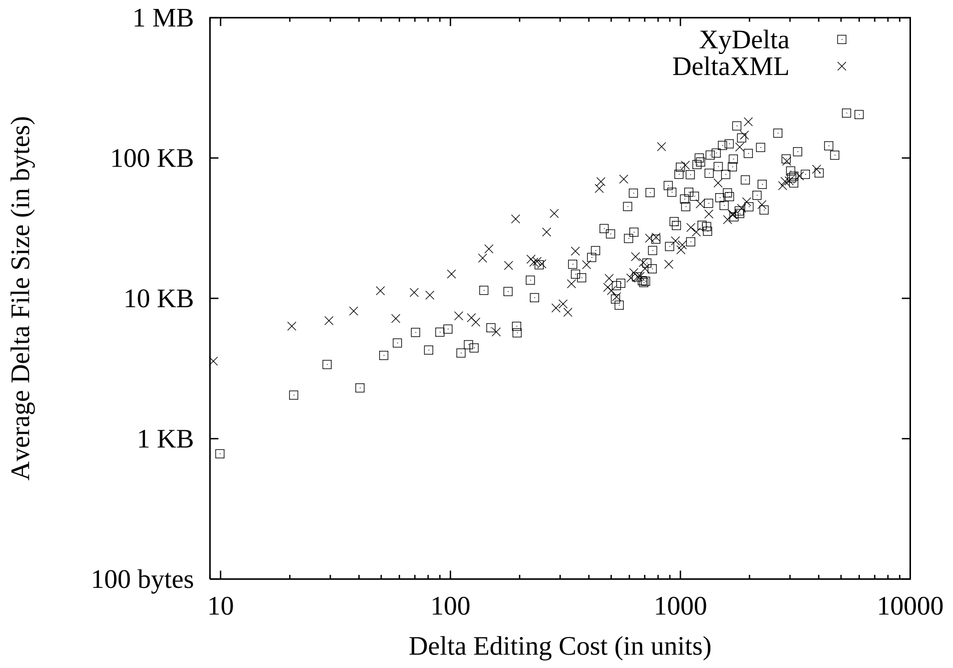

Figure 5.4: Size of the delta files

5.4 Diff calculation

In this section, we present an overview of previous works in this domain. The algorithms we describe are summarized in Figure 5.6 (page 101).

[6]Although fewer dots appear in the left part of the graph, they represent each the average over several hundred measures.

A *diff* algorithm consists in two parts: first it matches nodes between the two (versions of the same) document(s). Second it generates a document, namely a *delta*, representing a sequence of changes compatible with the matching.

The goal of our survey is to compare both the performance and the quality of several XML *diff* tools. In the next sections, we present experiments on the performance (see Section 5.5) and the quality (see Section 5.6) of the tools. In this section, we compare the tools based on the formal description of their algorithms (if available), and in particular we consider the upper-bound complexity and the minimality of the *delta* results.

Following subsections are organized as follows. First, we introduce the String Edit Problem. Then, we consider optimal tree pattern matching algorithms that rely on the string edit problem to find the best matching. Finally we consider other approaches that first find a "meaningful" mapping between the two documents, and then generate a compatible delta.

5.4.1 Introduction: The String Edit Problem

Longest Common Subsequence (LCS) In a standard way, the *diff* tries to find a minimum *edit script* between two strings. It is based on edit distances and the string edit problem [11, 65, 40, 106]. Insertion and deletion correspond to inserting and deleting a (single) symbol in a string. A cost (e.g. 1) is assigned to each operation. The string edit problem corresponds to finding an edit script of minimum cost that transforms a string x into a string y. A solution is obtained by considering the cost for transforming prefix substrings of x (up to the i-th symbol) into prefix subtrings of y (up to the j-th symbol). On a matrix $[1..|x|] * [1..|y|]$, a directed acyclic graph (DAG) representing all operations and their edit cost is constructed. Each path ending on (i, j) represents an edit script to transform $x[1..i]$ into $y[1..j]$. The minimum edit cost $cost(x[1..i] \rightarrow y[1..j])$ is then given by the minimal cost of these three possibilities:

$$
\begin{array}{l}
cost(deleteCharSymbol(x[i])) + cost(x[1..i-1] \rightarrow y[1..j]) \\
cost(insertCharSymbol(y[j])) + cost(x[1..i] \rightarrow y[1..j-1]) \\
cost(updateCharSymbol(x[i], y[j])) + cost(x[1..i-1] \rightarrow y[1..j-1])
\end{array}
$$

Note that for example the cost for $updateCharSymbol(x[i], y[j])$ is zero when the two symbols are identical. The edit distance between x and y is given by $cost(x[1..|x|] \rightarrow y[1..|y|])$, and the minimum edit script by the corresponding path.

The sequence of nodes that are not modified by the edit script (nodes on diagonal edges of the path) is a common subsequence of x and y. Thus, finding the minimal delta is equivalant to finding the "Longest Common Subsequence" (LCS) between x and y. Note that each node in the common subsequence defines a matching pair between the two corresponding symbols in x and y.

The space and time complexity are $O(|x| * |y|)$. This algorithm has been improved by Masek and Paterson using the "four-russians" technique [74] in $O(|x| * |y|/log|x|)$ and $O(|x| * |y| * log(log|x|)/log|x|)$ worst-case running time for finite and arbitrary alphabet sets respectively.

D-Band Algorithms In [81], E.W. Myers introduced a $O(|x| * D)$ algorithm, where D is the size of the minimum edit script. Such algorithms, namely *D-Band* algorithms, consist in computing cost values only close to the diagonal of the matrix. A diagonal k is defined by (i, j) couples with the same difference $i - j = k$, e.g. for $k = 0$ the diagonal contains $(0, 0), (1, 1), (2, 2), ...$ When using the usual "1 per node" cost model, diagonal areas of the matrix (e.g. all diagonals from $-K$ to K) contain all edit scripts of cost lower than a given value K. Obviously, if a valid edit script of cost lower than K is found to be minimum inside the diagonal area, then it must be the minimum edit script. When k is zero, the area consists solely in the diagonal starting at $(0, 0)$. By increasing k, it is then possible to find the minimum edit script in $O(max(|x| + |y|) * D)$ time. Using a more precise analysis of the number of deletions, [111] improves significantly this algorithm performance when the two documents lengths differ substantially. This *D-Band* technique is used by the famous **GNU diff** [47] program for text files.

5.4.2 Optimal Tree Pattern Matching

Serialized XML documents may be considered as strings, and thus we could use a "string edit" algorithm to detect changes. This may be used as a raw storage and raw version management, and can indeed be implemented using *GNU diff* that only supports flat text files. However, in order to support better services, it is preferable to consider specific algorithms for tree data that we describe next. The complexity we mention for each algorithm is relative to the total number of nodes in both documents. Note that the number of nodes is linear in the document's file size.

Previous Tree Models Kuo-Chung Tai [99] gave a definition of the edit distance between ordered labeled trees and the first non-exponential algorithm to compute it. Considering two documents $D1$ and $D2$, the time and space complexity is quasi-quadratic: $O(|D1| * |D2| * d(D1)^2 * d(D2)^2)$, where $d(D1)$ and $d(D2)$ represent the depth of the two trees. Zhang and Shasha [123, 124] proposed an algorithm with similar methods. The main difference is that it runs in a postorder traversal of the tree (child nodes are visited first, instead of preorder where parent nodes are visited first). The time complexity is $O(|D1| * |D2| * d(D1) * d(D2))$ and the space complexity is $O(|D1| * |D2|)$. This algorithm is used by *Logilab XML Diff* and *Microsoft XML Diff* that we present next. In the same spirit is Yang's [121] algorithm to find the syntactic differences between two programs.

In Selkow's variant [97], which is closer to XML, the LCS algorithm described previously is used on trees in a recursive algorithm. Considering two documents $D1$ and $D2$, the time complexity is $O(|D1| * |D2|)$.

MMDiff and XMDiff In [22], S. Chawathe presents an external memory algorithm *XMDiff* (based on a main memory version named *MMDiff*) for ordered trees in the spirit of Selkow's variant. Intuitively, the algorithm constructs a matrix in the spirit of the "string edit problem", but some edges are removed to enforce that deleting (resp. inserting) a node will delete (resp. insert) the subtree rooted at this node. More precisely, (i) diagonal edges exist if and only if corresponding nodes have the same depth in the tree (ii) horizontal (resp. vertical) edges from (x, y) to $(x + 1, y)$ exists unless the depth of node with prefix label $x + 1$ in $D1$ is lower than the depth of node $y + 1$ in $D2$. For *MMDiff*, the CPU and memory costs are quadratic $O(|D1| * |D2|)$. With *XMDiff*, memory usage is reduced but IO costs become quadratic.

Unordered Trees In XML, we sometimes want to consider the tree as unordered. The general problem becomes NP-hard [125], but by constraining the possible mappings between the two documents, K. Zhang [122] proposed an algorithm in quasi quadratic time. In the same spirit is **X-Diff** [109] from the project NiagaraCQ [27]. In these algorithms, for each pair of nodes from $D1$ and

$D2$ (e.g. the root nodes), the distance between their respective subtrees is obtained by finding the minimum-cost mapping for matching children (by reduction to the minimum cost maximum flow problem [122, 109]). More precisely, the complexity is $O(|D1| * |D2| * (deg(D1) + deg(D2)) * log(deg(D1) + deg(D2))$, where $deg(D)$ is the maximum outdegree (number of child nodes) of D. We do not consider these algorithms since we did not experiment on unordered XML trees. However, their characteristics are similar to *MMDiff* and both find a minimum edit script in quadratic time.

DeltaXML DeltaXML [44] is one of the nicest products on the market. It uses a similar technique based on longest common subsequence computations. It uses Wu [111, 81] *D-Band* algorithm to run in quasi-linear time. We believe[7] that the complexity is $O(|x| * D)$, where $|x|$ is the total size of both documents, and D the edit distance between them. The recent versions of DeltaXML support the addition of keys (either in the DTD or as attributes) that can be used to enforce correct matching (e.g. always match a *person* by its *name* attribute). DeltaXML also supports unordered XML trees.

Because Wu's algorithm is applied at each level separately, the result is not strictly minimal. Note that real-world experiments showed that the result is in general (90 percent) strictly minimal. To understand the algorithm, we provide here an example of non-minimal result that is obtained when diffing the following documents:

```
First Document:
<top>
<a><b>text</b><b>text</b><b>text</b><b>text</b></a>
<a><b>text</b></a>
<a><b>text</b></a>
</top>

Second Document:
<top>
<a><b>new text</b></a>
<a><b>updated text</b><b>text</b><b>text</b><b>text</b></a>
<a><b>updated text</b></a>
<a><b>updated text</b></a>
```

[7]The algorithm has not been published

```
</top>
```

The a and b nodes are mapped by pair in the order they appear, thus resulting in many operations (17 in total) to update their respective content. On the contrary, the minimal editing script consists in 6 operations.

Logilab XmlDiff In [67], Logilab proposes an Open Source XML diff. It offers two different algorithms. For large files, it uses Fast Match Edit Script [25] (from S. Chawathe and al.). As previously, this algorithm applies a LCS computation (using Myer's algorithm) at each level and for each label. Consequently, it runs in $O(l * |D1| * e)$, but does not find the minimal edit script (note that l is the number of node labels, and e the edit distance between the two documents).

The second algorithm is Zhang and Shasha tree-to-tree correction algorithm (mentioned previously). It finds a minimal edit script considering *insert* and *delete* operations according to Tai's model (see Section 5.2). More precisely, they use an extended version of [123, 124] that has been proposed in [13]. This version improve [123, 124] by adding a *swap* operation between a node and its next sibling. The complexity of this algorithm is quasi-quadratic, but the performance of the tool is slow[8]. Moreover, the implementation that we tested did not work for a large part of our test files. To represent changes, two options are supported: (i) XUpdate [114] language (ii) an internal format. We do not study the internal format since it is not XML based, and thus does not allow for further querying or native storage.

Microsoft XmlDiff In the same spirit, Microsoft recently proposed an XML Diff and Patch toolset [77]. It is free and the source code is freely available. The delta format is Microsoft *XDL*. This tool is in the spirit of *XML treediff* [56] developed by IBM and that was based on [38] and [123, 124]. Two diff algorithms are proposed. The first one is a fast tree-walking algorithm (in the same spirit as Fast Match). To be fast, they use a similar formula as *XyDiff* (see next) to limit the number of nodes visited during the tree walk.

The second algorithm is an implementation of Zhang and Shasha algorithm [123, 124]. As previously, note that the editing model considered here is Tai's model (see Section 5.2). Before any of the two algorithms is used, a preprocessing phase is applied to the documents, which consists in matching

[8]there may also be implementation issues

identical subtrees (based on their hash signature) in the spirit of *XyDiff* (see next). Matched nodes are then removed, and the algorithm choosen is then applied on the pruned tree. This improves significantly the performance, in particular for Zhang-Shasha algorithm. On the positive side, the result is that *move* operations are supported (based on the preprocessing matched nodes). On the negative side, the delta obtained is not minimal, as shown in the following example.

```
Source Document   | Target Document
                  |
<root>            |  <root>
  <a>             |    <a>
    <x/>          |      <x2/>
    <y/>          |      <x/>
  </a>            |      <y2/>
  <a>             |      <y/>
    <x/>          |    </a>
    <x2/>         |    <a>
    <y/>          |      <x/>
    <y2/>         |      <y/>
  </a>            |    </a>
</root>           |  </root>
```

Figure 5.5: Two versions of a document

Example Consider Figure 5.5 (page 97). The best choice is to *move* `x2` and `y2`. However, when considering *move* operations, finding the minimum edit script is in general NP-hard (see next). When *move* operations are ignored, some algorithms (e.g. Zhang-Shasha or *MMDiff*) find the minimum edit script. In this example, it consists in deleting `x2,y2` in the source document and then inserting `x2,y2` in the right place. Microsoft XmlDiff uses such an algorithm, but due to the preliminary pruning phase, the result is not minimal. More precisely, the two identical subtrees (`<a><x/><y/></a>`) in both documents are matched during the pruning phase, and as a consequence, a different edit script is found. It applies all modifications on the second subtree, and requires an additional *move* operations to swap the two subtrees. We consider in next section algorithms that support *move* operations.

5.4.3 Tree pattern matching with a *move* operation

The main reason why few *diff* algorithms supporting *move* operations have been developed is that most formulations of the tree diff problem are NP-hard [126, 24] (by reduction from the "exact cover by three-sets"). One may want to convert a pair of *delete* and *insert* operations applied on a similar subtree into a single *move* operation. But the result obtained is in general not minimal, unless the cost of *move* operations is strictly identical to the total cost of deleting and inserting the sutree.

LaDiff Recent work from S. Chawathe includes *LaDiff* [25, 24], designed for hierarchically structured information. It introduces a matching criteria to compare nodes, and the overall matching between both versions of the document is decided on this base. A minimal edit script (according to the matching) is then constructed. Its cost is in $O(n * e + e^2)$ where n is the total number of leaf nodes, and e a weighted edit distance between the two trees. Intuitively, its cost is linear in the size of the documents, but quadratic in the number of changes between them. Note that in terms of worst-case bounds, when the change rate is large the cost becomes quadratic in the size of the data. Since we do not have an XML implementation of LaDiff, we could not include it in our experiments.

XyDiff has been presented in Chapter 3 and in [35]. This tool is free and Open Source. *XyDiff* is a fast algorithm which supports *move* operations and XML features like the DTD ID attributes. Intuitively, it matches large identical subtrees found in both documents, and then propagates matchings. A first phase consists in matching nodes according to the key attributes. Then it tries to match the largest subtrees and considers smaller and smaller subtrees if matching fails. When matching succeeds, parents and descendants of matched nodes are also matched as long as the mappings are unambiguous. E.g., an unambiguous case is when two matched nodes have both a single child node with a given tag name. During the tree walk, the number of nodes visited is limited according to the "importance" (e.g. size) of the current subtree. This results in an upper bound (time and space) for the algorithm that is proved [35] to be no more than $O(n * log(n))$, where n is the size of the documents (e.g. total size of files). This algorithm does not, in general, find the minimum edit script.

5.4.4 Other Tools

Sun also released an XML specific tool named *DiffMK* [78] that computes the difference between two XML documents. This tool is based on the Unix standard *diff* algorithm, and uses a *list* description of the XML nodes.

In the same spirit, DecisionSoft proposes an Open Source XML diff program [43]. The program uses a linear representation of the XML document, i.e. the XML document is printed as text, and each printed line is considered as a node. Then, the Unix *diff* command is executed, and it finds which lines have been inserted or deleted. Consider the following document, in which we delete the first `author` subtree and the `phone` node in the second subtree:

```
<author>
    <name>Stefan Hellkvist</name>
</author>
<author>
    <name>Magnus Ljung</name>
    <phone/>
</author>
```

The resulting delta is as follows:

```
           <author>
DELETE       <name>Stefan Hellkvist</name>
DELETE     </author>
DELETE     <author>
             <name>Magnus Ljung</name>
DELETE       <phone/>
           </author>
```

On the serialized (flat) file, no changes are missed. However, this example shows that the tree structure of XML is not used. More precisely, we see that two `author` subtrees are merged by the deletion of two lines: `</author>` and `<author>`. Indeed, the notion of deleting a `</author>` line of text does not (in general) translate well into XML or tree operations. We consider that this result is not semantically “correct” in the context of our survey. These tools may be used for storage compression but can not be used for querying changes. Moreover, when we experimented with them, we found that the versions available at that time did not scale well to larger XML files. Thus, we did not include them in the speed and quality comparison.

5.4.5 Summary of tested *diff* programs

As previsouly mentioned, the algorithms are summarized in Figure 5.6 (page 101). The time cost given here (quadratic or linear) is a function of the data size, and corresponds to the case when there are few changes (i.e. $D << |x| + |y|$).

For GNU diff, we do not consider minimality since it does not support XML (or tree) editing operations. However, we mention in Section 5.6 some analysis of the result file size.

5.5 Experiments: Speed and Memory usage

As previously mentioned, our XML test data has been downloaded from the web. The files found on the web are on average small (a few kilobytes). To run tests on larger files, we used large XML files from DBLP [66] data source. We used two versions of the DBLP source, downloaded at an interval of one year.

The measures were conducted on a Linux system. Some of the XML diff tools are implemented in C++, whereas others are implemented in Java. Let us stress that we ran tests that show that these algorithms compiled in Java (Just-In-Time compiler) or C++ run on average at the same speed, in particular for large files.

For time reasons, we did not include Microsoft Xml Diff in Figure 5.7. Our experiments indicate that the performance of their Tree-Walking algorithm is similar to *XyDiff*, and the performance of Zhang-Shasha algorithm is (on average) similar to *DeltaXML*.

For space reasons, we didn't include *Logilab Tree-Walking* algorithm. It has roughly the same speed than *MMDiff*. A reason is that it uses a simple (and quadratic) implementation of the LCS at each level.

Let us analyze the behaviour of the time function plotted in Figure 5.7 (page 102). It represents, for each diff program, the average computing time depending on the input file size. On the one hand, *XyDiff* and *DeltaXML* are quasi-linear, as well as *GNU Diff*. On the other hand, *MMDiff* increase rate corresponds to a quadratic time complexity. When handling medium files (e.g. hundred kilobytes), there are orders of magnitude between the running time of linear vs. quadratic algorithms.

Program Name	Author	Time	Memory	Moves	Minimal Edit Cost	Notes
fully tested						
XyDiff	INRIA	linear	linear	yes	no	
DeltaXML	DeltaXML.com	linear	linear	no	no	
MMDiff	Chawathe and al.	quadratic	quadratic	no	yes	tests with our implementation
XMDiff	Chawathe and al.	quadratic	linear	no	yes	quadratic I/O cost tests with our implementation
GNU Diff	GNU Tools	linear	linear	no	-	no XML support (flat files)
XML Diff (Fast) (ZhangShasha)	Logilab Logilab	quadratic quadratic	quadratic quadratic	no no	no yes	
XML Diff (Fast) (ZhangShasha)	Microsoft Microsoft	linear quadratic	linear quadratic	yes yes	no no	(Pruning phase first)
not included in experiments						
LaDiff	Chawathe and al.	linear	linear	yes	no	criteria based mapping
XMLTreeDiff	IBM	quadratic	quadratic	no	no	(see Microsoft XML Diff)
DiffMK	Sun	quadratic	quadratic	no	no	no tree structure
XML Diff	Dommitt.com					we were not allowed to discuss it
XML Diff	DecisionSoft					no real XML support (flat text)
Constrained Diff	K. Zhang	quadratic	quadratic	no	yes	-for unordered trees -constrained mapping
X-Diff	Y. Wang, D. DeWitt, Jin-Yi Cai (U. Wisconsin)	quadratic	quadratic	no	yes	-for unordered trees -constrained mapping

Figure 5.6: Quick Summary

For *MMDiff*, memory usage is the limiting factor since we used a 1Gb RAM PC to run it on files up to hundred kilobytes. For larger files, the computation time of *XMDiff* (the external-memory version of *MMDiff*) increases significantly when disk accesses become more and more intensive.

In terms of implementation, *GNU Diff* is much faster than others because it doesn't parse or handle XML. This should be linked to experiments on *XyDiff* that showed that 90 percent of the time is spent in the XML parser. This makes *GNU Diff* very performant for simple text-based version management schemes.

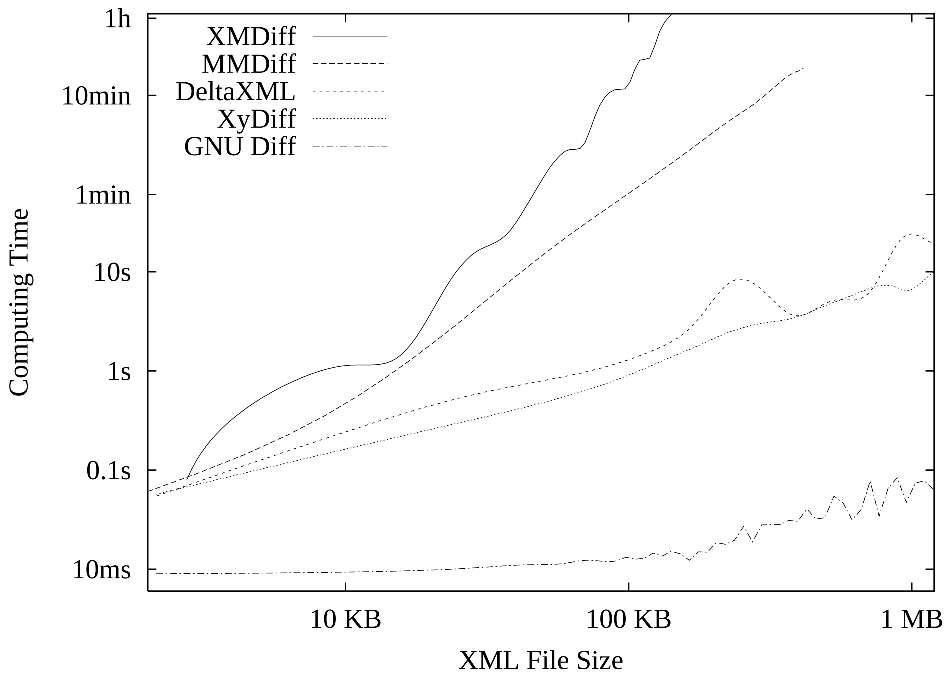

Figure 5.7: Speed of different programs

A more precise analysis of *DeltaXML* results is depicted in Figure 5.8 (page 103). Its shows that although the average computation time is linear, the results for some documents are significantly different. Indeed, the computation time is almost quadratic for some files. We found that it corresponds to the worst case for *D-Band* algorithms: the edit distance D (i.e. the number of changes) between the two documents is close to the number of nodes N. For instance, in some documents, 40 percent of the nodes changed, whereas in other documents less than 3 percent of the nodes changed. This may be a slight disadvantage for applications with strict time requirements, e.g. computing the diff over a flow of crawled documents as in NiagaraCQ [27] or Xyleme [84]. On the contrary, for

MMDiff and *XyDiff*, the variance of computation time for all the documents is small. This shows that their average complexity is equal to the upper bound.

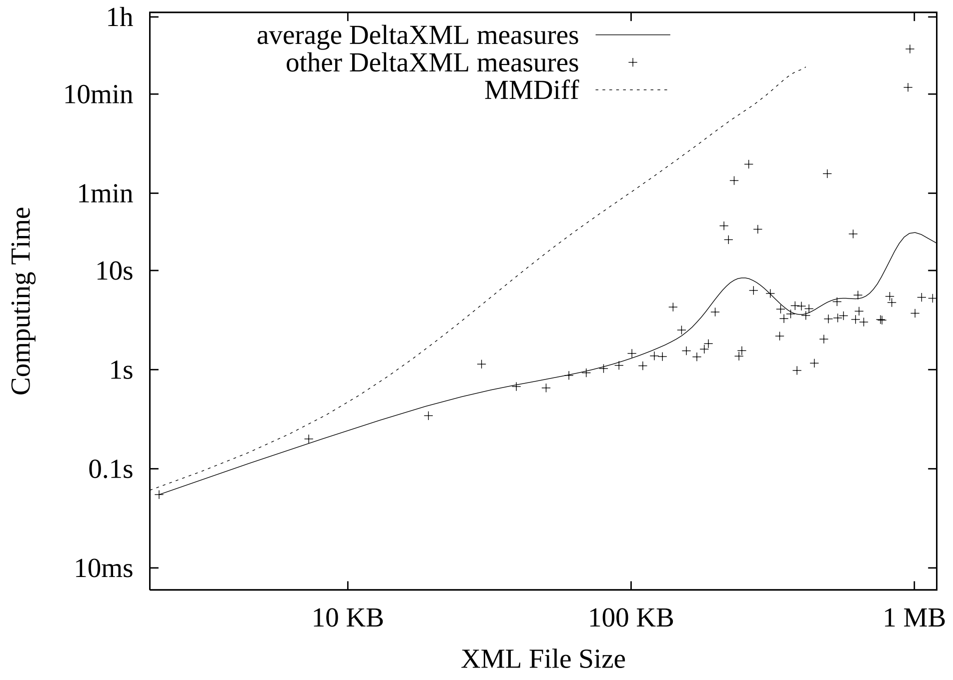

Figure 5.8: Focus on DeltaXML speed measures

5.6 Quality of the result

The "quality" study in our benchmark consists in comparing the sequence of changes generated by the different algorithms. We used the result of *MMDiff* and *XMDiff* as a reference because these algorithms find the minimum edit script. Thus, for each pair of documents, the quality for a diff tool (e.g. DeltaXML) is defined by the ratio

$$r = \frac{C}{C_{ref}}$$

where C is the delta edit cost and C_{ref} is *MMDiff* delta's edit cost for the same pair of documents. A quality equals to one means that the result is minimum and is considered "perfect". When the ratio increases, the quality decreases. For instance, a ratio of 2 means that the delta is twice more costly than the minimum delta. In our first experiments, we didn't consider *move* operations. This was done by replacing for *XyDiff* each *move* operation by the corresponding pair of *insert* and *delete* . In this case, the cost of moving a subtree is identical to the cost of deleting and inserting it.

In Figure 5.9 (page 104), we present an histogram of the results, i.e. the number of documents in some range of quality. *XMDiff* and *MMDiff* do not appear on the graph because they serve as reference, meaning that all documents have a quality strictly equal to one. *GNU Diff* do not appear on the graph because it doesn't construct XML (tree) edit sequences. The results of *Microsoft Diff* and *Logilab Diff* do not appear on the graph because they use a different change model (Tai's operations).

These results in Figure 5.9 show that:

- (i) DeltaXML: For most of the documents, the quality of *DeltaXML* result is perfect (strictly equal to 1). For the others, the delta is on average thirty percent more costly than the minimum.

- (ii) XyDiff: Almost half of the deltas are less than twice more costly than the minimum. The other half costs on average three times the minimum.

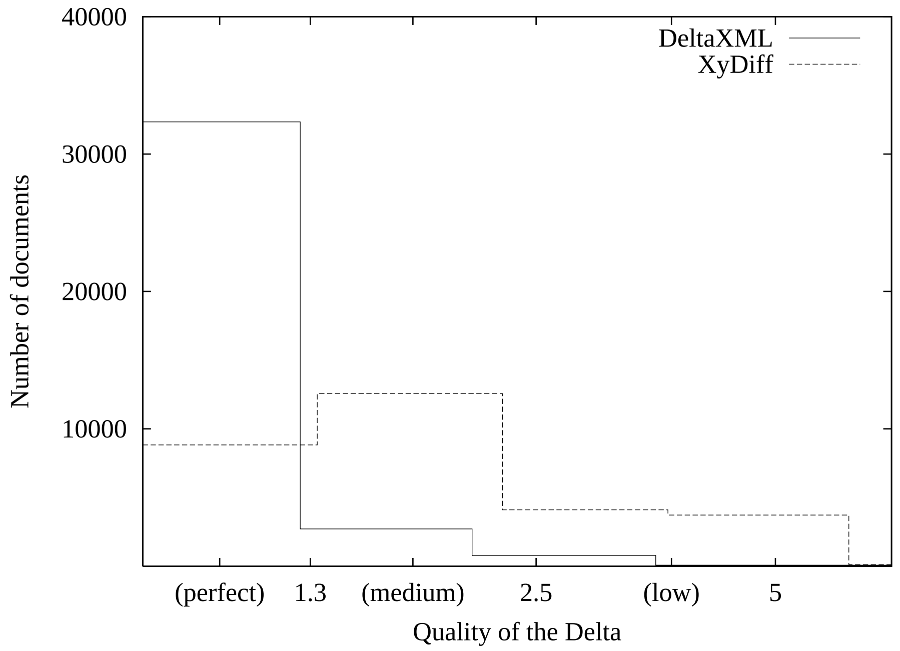

Figure 5.9: Quality Histogram

Delta files size (without move) In terms of file sizes, we also compared the different delta documents, as well as the flat text result of *GNU Diff*. The result *diff* files for *DeltaXML*, *GNU Diff* and *XyDiff* have on average the same size. The result files for *MMDiff* are on average twice smaller (using a *XyDelta* representation of changes). The result files of *Microsoft Diff* are also on average twice smaller. The reason is often the use of Tai's change model which results in shorter edit scripts. Another reason is the support of *move* operations.

Using "move" We also conducted experiments by considering *move* operations and assigning them the cost 1. Intuitively this means that *move* is considered cheaper than deleting and inserting a subtree, e.g. moving files is cheaper than copying them and deleting the original copy. Only *XyDiff* and *Microsoft XmlDiff* detect *move* operations. On average, *XyDiff* performs better and becomes better than the reference for fifteen percent of the documents, and in particular for large files. The results of *Microsoft Diff* often contain *move* operations that improve their accuracy: with limited human checking of the results we found that, for large files, half of the edit scripts are smaller (about a half) than the reference.

Finally, note that this quality measure focuses on the minimality of results. In some applications, the semantics of the results is more important. But the semantic value can not be easily measured. An interesting aspect is the support of (semantic) matching rules by some programs (DeltaXML, XyDiff). More work is clearly needed in the direction of evaluating the semantic quality of results. We also intend to conduct experiments on *LaDiff* [25] which is a good example of criteria-based mapping and change detection.

5.7 Conclusion

In this Chapter, we described existing works on the topic of change detection in XML documents.

We first presented recent proposals for representing changes, and compared their features through analysis and experiments. We believe that more work is needed to propose a framework for version management and querying changes according to the proposed change languages. Persistant identifiers (as in XyDelta) are an important aspect. As one should expect, languages focusing on edit operations (e.g. XUpdate, XyDiff) are slightly more compact than languages summarizing the document (e.g. DeltaXML, XDL). But the latter are more easily integrated in simple applications, such as monitoring. Important features should also be considered, like the support of *move* operations (e.g. XDL, XyDelta) or backward deltas. At the cost of little improvements, the languages presented here are close to be equivalent. It would be interesting to have a common standard.

The second part of our study concerns change detection algorithms. We compared two main approaches, the first one consists in computation of minimal edit scripts, while the second approach relies on meaningfull mappings between documents. We underlined the need for semantical integration in the change detection process. The study and experiments presented show (i) a significant quality advantage for minimal-based algorithms (MMDiff, DeltaXML and Microsoft Diff (ZhangShasha)) (ii) a dramatic performance improvement with linear complexity algorithms (GNU Diff, XyDiff, Microsoft TreeWalking, DeltaXML[9]).

On one hand, only MMDiff finds the exact minimal edit script, but it does not scale to large files (e.g. 1Mb). Moreover, only Tree-Walking algorithms (GNU Diff, XyDiff, Microsoft TreeWalking) always run in linear time, but the quality of their results is lower. DeltaXML runs on average in linear time, but the cost may be quadratic for some files.

On the other hand, DeltaXML and Microsoft Zhang-Shasha offer good compromises to find high-quality deltas. Both run on average in quasi linear time, although they may take longer for some large files. The main difference between them is the change model used. The one used in DeltaXML may be better for some XML documents, whereas the change model used in Microsoft Diff often results in smaller deltas.

We also noted that flat text based version management (GNU Diff) still makes sense with XML data for performance critical applications.

Although the problem of “diffing” XML (and its complexity) are better and better understood, there is still room for improvement. In particular, *diff* algorithms could take better advantage of semantic knowledge that we may have on the documents or may have infered from their histories or their DTD.

[9]linear time on average, but significantly more for some files

Chapter 6

Monitoring XML

Abstract *In previous chapter, we presented state-of-the art tools for detecting and representing changes in XML. This work is in the context of change-control for semi-structured data. In this chapter, we focus on the 'control' part: we present algorithms and tools for continuous queries on XML data that have been developed in the context of the Xyleme project [117]. These tools have been transfered to the industry [119].*

This work has been conducted with the help of Benjamin Nguyen, Serge Abiteboul and Mihai Preda. In particular, the subscription system has been presented in [84] by Benjamin Nguyen. Benjamin Nguyen was in charge of the system's architecture, and implemented the first prototypes with the help of Jeremy Jouglet and Mihai Preda. My contributions where the definition, design and implementation of the Alerters.

6.1 Introduction

Our system is a scalable architecture that enables the notification of users (or applications) when specific events are detected. The important features of this system are:

- Management of subscriptions by a large number of users
- Acquisition of the XML data to be processed
- Processing of user queries on the data, with extensive sharing of redundant computations among various subscriptions.

- Preparing and sending notification reports to users (or applications) depending on their interests and their subscriptions

The *Alerters* are modules that process documents in order to detect specific small-grain events, namely atomic events, such as the presence of a certain keywords. The conjunction of such events leads to the trigger of notification reports, namely complex events, as defined by the users.

The definition of the Alerters consists in three important aspects:

- Defining the service. This means defining with kind of atomic events will be detected. For each kind of events, for instance *keywords detection*, the alerters will receive the different events registrations depending on users subscriptions. For instance, a user subscription may contain different atomic events and may lead to the registration of three different words: *inria*, *Xyleme* and *XML*.

- The design and implementation of the algorithms that implement the event detection of a data flow. The alerters will have to process hundred of documents per second.

- The definition of a scalable architecture in order to: (i) increase the processing power and speed of alerters by adding more machines, (ii) maintain the whole detection chain up-to-date and consistent with concurrent (un)subscriptions and a permanent data flow.

In the context of the Xyleme project, our goal was to process HTML and XML documents found on the Web.

Given the size of the Web, it was necessary to be able to process several millions of documents per day. Whereas the services that we provide are not new, this context lead to very strict performance requirements that made it a challenging problem. While we do not need to have strict real-time processing (i.e. with very low latency), the average speed of the alerters has to match the crawling speed. The word *streaming* is often used to describe this problem of processing a flow of data in this context [96]. Intuitively, it means that the alerters have to process the flow of data and not slow it down.

One aspect of the architecture is the extensive use of asynchronous communications in the system. For instance, the alerters receive documents in a asynchronous way: documents are sent by burst transfers and no reply is needed. In a similar way, the alerters send their alerts to the next module in the chain (the Notification Processor) in an asynchronous way: alerts are sent using burst transfers and no reply is expected. The synchronous transfers enable a large gain on network bandwidth since raw bandwidth is not a limiting factor, while network calls latency may be a limiting factor. Also it implies that a module does not slow down the other modules as long as its average speed for handling documents matches the average speed of its input.

We present briefly in this chapter the two algorithms that we implemented and that represent the main aspects of events detection in our system. Namely, we present an algorithm to detect keywords (and keywords sequences) and an algorithm to detect simple path expressions (in the spirit of XPath). We also present a brief state-of-the art on the topic of processing XPath queries.

Finally, we mention a real-worlds application of the system: the *Copy Tracker* that has been developed by Mihai Preda using the notification system. This is a critical use-case of our system since it is targeted to the final customers.

6.2 Alerts on keywords

Detecting keywords on pages is a basic feature of any notification system. However, the problem becomes more complex when considering millions of users (i.e. millions of keywords), and a few hundred documents per second.

For engineering reasons, and given the experiments for the URL plug-in, we decided to also use an hash table to implement this plug-in. The look-up of a given word consists simply in a look-up in the hash table. The document is read, and each word is processed in the order they appear. Note that for a word that appears several times in the document, a look-up in the hash table is computed as many times.

The Sequence Problem An interesting point was the detection of keywords sequence. Consider for instance, `Air France`. A possibility to implement the detection is as follows. As described in [84], our system detects conjunctions of events. Thus, it can detect that a page contains the two words `Air` and `France`. It is possible, using some post-processing to detect among those pages that contain the two words which of them contain the word sequence `Air France`. However, the performance and complexity study conducted by Benjamin Nguyen in [84, 83] seemed to indicate that this was not an optimal choice as it would increase the number of detected events, which increases dramatically the cost of finding the corresponding subscriptions.

The Sequence Algorithm We developed a specific algorithm for detecting word sequences. More precisely, we consider the detection of a sequence of consecutive words $W_1, W_2, w_3, ...w_n$ in the document. We implemented it as follows:

- a `WordsMemory` structure records the n_{max} latest words of the document. n_{max} is typically from 5 to 100.
- The words sequence as registered in the hash table as one large word, including space characters. For instance `"Air France"`.

Then, when the document is read, each possible subsequence (with length up to n_{max}) is tested against the hash table. This results in $s * n_{max}$ look-ups in the hash table, where s is the number of words in the document, and n_{max} the maximum number of words allowed for each sequence.

When n_{max} grows, it becomes inefficient to test all sequences up to that length. We introduced an optimization as follows:

1. For each subscription $w_1, ..., w_n$, the subsequence $w_1, ..., w_j$, where $j < n$ are marked “interesting”. They are added to the hash table (associated with a null event).
2. Consider that a word w is read from the document. A look-up for w in the hash table is processed. This mean that we look-up for an event corresponding to the sequence of length i where $i = 1$, ending at word w. If some event (even a null event) is found, we increase i and we loop. This is done until $i = n_{max}$ or until no event is found for some value j of the length of the sequence. Then, we continue processing the document by reading next word w'.

For instance, consider the sequence `...avion Air France...` in the document. We look-up for `avion` in the hash table. No event is found. We read the next word. We look-up for `Air` in the hash table. No event is found. We read the next word. We look-up for `France` in the hash table. A null event is found. We look-up for `Air France`. The corresponding event is found (and not null).

We note it. We look-up for `avion Air France`. No event is found. We read the next word.

Thus, the worst-case upper bound remains identical in term of s and n_{max}, but the average computation time is greatly reduced.

Remark Note that the management of deletions of subscriptions requires to have a usage counter for each string in the hash table. Interesting, the structure that is then represented in the hash table is somehow similar to a dictionary (where node granularity is an entire word), of the reversed sequences that have been subscribed. We will see next that this can be compared to some XPath filtering algorithm which consists in reversing path expressions.

6.3 An application: the Copy-Tracker

In this section, we briefly describe a typical application that uses the Alerters. This application is named *Copy Tracker*, and has been developed by Mihai Preda, from [119].

The context of the application was to detect illegal copies of News wires on the Internet. The goal of our system was, given a News wire, to retrieve all copies of that wire found on the Internet. To do so, we use the Xyleme Crawler to read pages from the Web, and the Alerter to detect events on these pages. The Copy-Tracker consists in finding for each news wire the specific events that can be used to detect the copies on the Web. This is done by finding a *signature* for each news wire. The signature consists in a set of words that (we believe) are specific to that particular text. The alerters are used to detect all pages on the Web containing these words, i.e. matching the signature.

The *copy-tracker* application process consists in two steps.

The first step consists in finding the signature of the document. The signature is a set of discriminant words in the News wire. These are typically infrequent words that are frequently used in that precise document. More precisely, for each word we compute a score that is the ratio between its frequency of use in the document versus its frequency on the Web.

$$r = \frac{f_{document}}{f_{Web}}$$

We select the n words with highest ratio. n is typically 4 to 7. When n is to low, many Web pages may be detected that are not the expected News wire. On the other hand, when n is too high, some versions of the News wire that have been slightly modified may not be detected. The frequency of words on the Web, f_{Web} is itself computed using the Alerters. This was done once during a limited time-frame experiment. We registered each word from the dictionary (to avoid spelling mistakes), and for each word, the alerters reported each time a document appears which contains it.

The second step consists in finding on the Web all documents that match this signature. To do so, we register a continuous query, to the subscription system, that reports all documents which contain all selected words. Then, this query is evaluated on documents that are loaded from the Web. An important issue in that system, that we do not consider here, is that news pages change frequently on the Web. Thus, the crawling strategy must be very efficient to rapidly discover *new* news pages and crawl them before they change or disappear. This is considered in Chapters 7 and Chapter 8.

This application shows a typical usage of the Alerters to detect rapidly specific documents on the Internet. It is a real-world industrial applications, that targets directly the final customer.

6.4 XML Alerts

In this section, we present an algorithm that we implemented to detect specific path expressions in XML documents. There is abundant work in that area, and more precisely on the topic of processing XPath queries.

First, we present briefly what is a path expression as proposed by XPath. Then, we present the state of the art for processing XPath expressions. Finally, we present our algorithm.

6.4.1 A brief introduction to XPath

Consider an XML document. A simple path expression is a sequence of descending axes (e.g. child-of or descendant-of) that is used to retrieve nodes in the document. For instance, consider the expression:

```
/catalog/product//price
```

It starts on the document's root node, that should be named `<catalog>`. If not, the result is empty. Then, it takes all child nodes of `<catalog>` that are named `<product>`, and for each of them, retrieves all the nodes named `<price>` in their subtree (i.e. descendant nodes).

In that spirit, XPath [116] is a language for addressing parts of an XML documents. An XPath expression starts from a node that is named the context node. It retrieves a list of nodes, each of them matching the path expression. By default, the context node is the root of the document. If the context is a list of nodes, the XPath is applied to each context node, and the results are merged. Details omitted.

XPath does significantly more than simple path expression in several ways:

- It allows not only descendant axes (e.g. `child-of`(noted `/`), or `descendant-of` (noted `//`), or `next-sibling`), but also ascending axes, such as `parent`, `ascendant-of` or `previous-sibling`.

  A typical example is:

  ```
  //price/parent::cd/parent::product
  ```

  It retrieves all parent nodes of `product` nodes.

- It allows the evaluation of filtering, such as `name="computer"`, including a full algebra on numbers and strings. As we will see later, this algebra (including string concatenation) and tests are sufficient to make XPath evaluation very costly in time and space. A typical example is:

  ```
  //product/cd[price<9.99]
  ```

  It retrieves all product nodes which have a `price` child. Then, the text node below the price is compared with 99, and only the node giving true results are kept.

- It allows to *"join"* several XPath expressions. The join is based on node identity and represented by `[]`. For instance consider `P1[P2]/P3`. The expression `P1` retrieves a set of context nodes. The expression `P2` is applied on each of them. Only the nodes for which the result is non-empty are kept. Then, `P3` is applied on the remaining nodes. Following example retrieves, among all products, the name of those whose price is lower than 99.

```
//product/cd[price<9.99]/name
```

In that case, P1 is `//product/cd`, P2 is `price<"99"`, and P3 is `name`.

The formal definition of XPath gives a way to evaluate XPath expressions. However, we see next that real-world implementations of XPath should use other algorithms.

6.4.2 Processing XPath queries: state of the art

Algorithms for processing of XPath queries should be considered in two categories: (i) the regular ones, evaluating the query using knowledge about the entire document (typically a DOM model), and (ii) streaming XML, i.e. evaluation query with a single pass on the XML document, and limited memory usage (typically a pushdown stack).

DOM Model Recent work on that topic is presented in [51]. First, they show through experiments that the current implementations of XPath in XALAN [112], XT [33] and Microsoft Internet Explorer 6 have an exponential cost (in the length of the query), even for very simple documents and queries. [51] proposed a polynomial algorithm for evaluating full XPath, and a linear time algorithm for a subset of XPath. Intuitively, this subset of XPath contains all XPath axes, but no arithmetical or string operations. The linear time algorithm runs in $O(N * D)$ where N is the size of the query and D the size of the data. In [52], the polynomial algorithm is improved, and a practically relevant fragment of XPath is defined for which a further form of query evaluation is possible. The main drawback of these algorithms is that they rely on a DOM model, i.e. the document has to be loaded in memory [1].

Streaming XML It is also very common to consider some algorithms and applications in the context of *streaming XML*. Streaming XML typically relies on the SAX parser model [93]. It assumes that the document is traversed once. While some streaming algorithms use no memory at all, most algorithms use a pushdown stack [96]. In [53], a streaming version of an XPath algorithm is proposed. It has been implemented has part of the XML-TK [115] project. It consider the descendant paths, and does not consider string (and numerical) algebra. It also

[1]Another possibility is to use a Persistent DOM implementation and navigate on disk

does not consider joins. The main idea is that path expressions can be processed using a Non-Deterministic State Automata (with no stack). Their algorithms constructs lazily the corresponding deterministic state automata on the fly, while the XML document is processed. The construction is lazy in that it only constructs the parts of the automata that are reached while reading the document. While the deterministic automata size is exponential in the worst-case (i.e. to handle all possible documents), the automata size remains linear (bounded by the data guide) when a single document is processed. Thus, the worst-case cost for processing a single document is in $O(N * D)$, where N is the size of the query, and D the size of the data. Moreover, it is possible to process several queries at a time by composition of their automata. In that case, the computing cost is no more than $O(D * \sum_q N_q)$.

Ascending Axes As we have seen, several algorithms consider only descending axes of XPath, for instance *child*, *nextSibling* and not *parent*, *previousSibling*. However, [86] proves that this can be done without loss of generality. Indeed, they proposed two algorithms that transform path expression (without joins) into descending only path expressions. A first algorithm runs in exponential time (in the length of the path) and transforms a path expression into a descending path expression, with no joins. To do so, there is a restriction on the ascending axes of the first path expression, i.e. they can not use *ancestor* axes but only *parent*. A second algorithm runs in linear time and constructs a path expression, with as many joins as ascending axes in the source.

Filtering XML In [21], an index structure is proposed that is used to process XPath expressions in a streaming fashion. Their work relies on decomposing tree patterns into collections of substrings (i.e. simple path expressions), and indexing them. A closely related work is XFilter [10].

6.4.3 Our algorithm for processing simple path expressions

Our work is also in the spirit of XFilter [10]. The main difference with XPath is that we do not return the set of nodes that match the path expression. Given an XML document, and a set of path queries, we only return a list of those path expression that have had a non-empty result for that document. Other restrictions to XPath are:

(i) we only consider descending axes,

(ii) we do not consider an algebra for processing numbers and strings,

(iii) we do not allow the presence of several branches (denoted by `[]` in XPath), that lead to computing joins.

Intuition Our algorithm works by reversing the path expressions. Then, the document is read in a streaming fashion, and nodes are processed in a postfix order. For instance, a node is processed when its closing XML tag is read, and not when its opening tag is read. Thus, the parent and ancestors of a node are processed after each node. By checking ancestor nodes against the reversed path expressions, we can detect the corresponding events. A detailed description is given below:

Description of the algorithm Consider a path expression P, composed of a sequence of element names $e_1, ..., e_n$. The document is read in a streaming fashion, in postfix order, i.e. child nodes are always processed first. Let l be the level of a node, i.e. the node's depth starting from the root of the document. We maintain an array that stores, for each possible level, a set of element's position in the path expression. These positions start from the right of the path expression, i.e. the path expression is considered in reversed order. They correspond to the parts of the path expression that have already been detected (the right part), and point to the part that has to be detected (the left part). For instance, at the beginning, all lists contain a single pointer to e_n. If an element of type e_n is found at some node n, then, a pointer to e_{n-1} is added to list corresponding to the previous level $l-1$. Again, note that the algorithm rely on the postfix order of nodes, e.g. the parent node of n will be read after n was read. This is done until e_1 is detected, which implies that the path expression has a non-empty result to that document.

We handle differently the two operators / and //. For a path denoted by *.../a//b*, once b has been detected, a should be tested against all ancestor nodes. First, a pointer to a is stored at the level corresponding to the parent node p of b. When node p is read, we check if p is an element node of name a. If so, the rest of the path expression is stored at his parent level, and so on. If not, the same pointer to a is stored at the parent level. If the path expression was *.../a/b*, and the test failed, then the pointer to a is thrown away.

An important aspect to note is that all elements of path expressions are indexed in a shared hash table, in order to retrieve at little cost matching path expressions.

From there, it is easy to extend the algorithm to work on several path expression while reading the document only once. The precise algorithm is given in Figure 6.1

Complexity Analysis Let D be the number of nodes in the document. For each node, we have to find all path expressions ending with that node. The cost is constant since this is a look-up to a hash table. Then, we have to test the node again all path expression pointers stored at the current stack level, and possibly move them to the parent level. In the worst case, there is at each level of the stack, a pointer to each position of each registered path expression. Thus, the worst-case cost is in $O(D * d * \sum_q N_q)$, where D is the size of the document, d the depth of the document, and N_q the length of each query q.

Experiments The algorithm has been implemented as part of the Xyleme project [117], and is now used in a production system in Xyleme [119]. Thus, it has been tested against millions of XML documents loaded from the Web.

6.5 Conclusion

We have developed an algorithm for processing simple XPath queries that is tailored to the needs of the Xyleme system and runs in quasi linear time.

Next, we plan to improve our algorithm as follows. The possibility to support the XPath operator `[]` (i.e. joins based on nodes identities) may be added at the subscription level where several atomic events are joined. However, to do so, it would be necessary to do joins based on node identity, and thus it is necessary to modify our algorithm so that it return the list of nodes matching the various path expressions. When this is done, according to Olteanu and al. work [86], we could support descending axes *(i)* using a preprocessing phase for path queries.

In the context of Active XML [1], we have also conducted research on the topic of query evaluation on distributed and replicated data [2]. This work is not detailed here.

More experiments are also needed.

```
% Input:
%    Document D,
%       (all nodes <n> in postfix order)
%    A Set of path Expressions P1, ..., Pq
%
% Output:
%    Set of path expressions a1....ak
%    where a_i is in (P1...Pq)
%

Stack s ;
ReturnResult r;

for each node <N>
  % in postfix order

  let l be the level of N

  find all path expressions ending with N
  add them to s, at level (l-1)

  for each element <e> of s at level (l)

    if N is equal to e
      if <e> is the first element
             of path expression P
      then add P to r
      else add previous element of e
               to s at level (l-1)

    else
      if the operator is "e/"
         then forget e
      if the operator is "e//"
         then move e to s at level (l-1)

    endif
  end for
end for
```

Figure 6.1: Streaming Computation of Simple Path Expressions

Part II

Archiving the french Web

Introduction

In the first part of this thesis, our focus was change-control at the microscopic scale, and in particular changes inside XML documents. We presented algorithms and systems to analyze the elements and components of semi-structured data, as well as detecting and representing changes.

In this part, our focus will be changes at the macroscopic scale. More precisely, we are interested in finding and managing data and documents of interest from the web. The goal is to set-up foundations for systems that manage historical data, found anywhere on the web, possibly in the context of some specific application.

We will in particular see the notion of page importance, namely PageRank, that is an essential aspect towards efficient approaches for data discovery on the Internet.

In Chapter 7, we propose a new algorithm that computes online the importance of web pages. In Chapter 8, we present our work in the context of an interesting application: the archiving of the web by national libraries, and more precisely the archiving of the French web by the French national library (BnF).

Chapter 7

On-line Computation of page importance

Abstract *The computation of page importance in a huge dynamic graph has recently attracted a lot of attention because of the web. Page importance, or page rank is defined as the fixpoint of a matrix equation. Previous algorithms compute it off-line and require the use of a lot of extra CPU as well as disk resources (e.g. to store, maintain and read the link matrix). We introduce a new algorithm OPIC that works on-line, and uses much less resources. In particular, it does not require storing the link matrix. It is on-line in that it continuously refines its estimate of page importance while the web/graph is visited. Thus it can be used to focus crawling to the most interesting pages. We prove the correctness of OPIC. We present Adaptive OPIC that also works on-line but adapts dynamically to changes of the web. A variant of this algorithm is now used by Xyleme.*

We report on experiments with synthetic data. In particular, we study the convergence and adaptiveness of the algorithms for various scheduling strategies for the pages to visit. We also report on experiments based on crawls of significant portions of the web.

This work has been published in [7], and an extended abstract has been published in [5]. It has been conducted with Serge Abiteboul and Mihai Preda. In particular, the original idea of such an algorithm is from Mihai, as well as the implementation and experiments on a large scale web Crawler. Serge helped him "fix" the algorithm (make it compute the correct PageRank) and proved the correctness. My contributions are as follows:

- Although not leading, I participated in the early works on the algorithm

- I participated in the proof of correctness of the algorithm and in tuning it
- I implemented the algorithms and conducted the experiments on synthetic data
- I conducted the research on dynamic graphs (model and experiments)
- Together with Luc Segoufin, we further formalized the PageRank computation over the graph of the web, and explained issues related to a-periodicity and strong connectivity of the graph.

7.1 Introduction

An automated web agent visits the web, retrieving pages to perform some processing such as indexing, archiving, site checking, etc., [9, 49, 94]. The robot uses page links in the retrieved pages to discover new pages. Observe that all pages on the web do not have the same importance. For example, Le Louvre homepage is more important that an unknown person's homepage. Page importance information is very valuable. It is used by search engines to display results in the order of page importance [49]. It is also useful for guiding the refreshing and discovery of pages: important pages should be refreshed more often[1] and when crawling for new pages, important pages have to be fetched first [31]. Following some ideas of [62], Page and Brin proposed a notion of page importance based on the link structure of the web [16]. This was then used by Google with a remarkable success. Intuitively, a page is important if there are many important pages pointing to it. This corresponds, for instance, to the intuition of importance for research articles: a paper is important if it is referenced by many other (important) papers. This leads to a fixpoint computation by repeatedly multiplying the matrix of links between pages with the vector of the current estimate of page importance until the estimate is stable, i.e., until a fixpoint is reached.

The main issue in this context is the size of the web, billions of pages [15, 92]. Techniques have been developed to compute page importance efficiently, e.g., [55]. The web is crawled and the link matrix computed and stored. A version of the matrix is then frozen and one separate process computes off-line page importance, which may take hours or days for a very large graph. So, the core of the technology for the off-line algorithms is fast sparse matrix multiplication (in

[1]Google [49] seems to use such a strategy for refreshing pages; Xyleme [119] does.

particular by extensive use of parallelism). This is a classical area, e.g., [100]. The algorithm we propose computes the importance of pages on-line, with limited resources, while crawling the web. It can be used to focus crawling to the most interesting pages.

The algorithm works as follows. Intuitively speaking, some "cash" is initially distributed to each page and each page when it is crawled distributes its current cash equally to all pages it points to. This fact is recorded in the history of the page. The importance of a page is then obtained from the "credit history" of the page. The intuition is that the flow of cash through a page is proportional to its importance. It is essential to note that the importance we compute does not assume anything about the selection of pages to visit. If a page "waits" for a while before being visited, it accumulates cash and has more to distribute at the next visit. In Sections 7.2 and 7.3, we present a formal model and we prove the correctness of the algorithm.

In practice, the situation is more complex. Consider the ranking of query results. First, the ranking of result pages by a search engine should be based on factors other than page importance. One may use criteria such as the occurrences of words from the query and their positions. These are typically criteria from information retrieval [101] that have been used extensively since the first generation of search engines, e.g. [9]. One may also want to bias the ranking of answers based on the interest of users [88, 20]. Such interesting aspects are ignored here. On the other hand, we focus on another critical aspect of page importance, the variations of importance when the web changes.

The web changes all the time. With the off-line algorithm, we need to restart a computation. Although techniques can be used to take into account previous computations, several costly iterations over the entire graph have to be performed by the off-line algorithm. We show how to modify the on-line algorithm to adapt to changes. Intuitively, this is achieved by taking into account only a recent window of the history.

Several variants of the adaptive on-line algorithm are presented. An implementation of one of them is actually used by the Xyleme crawlers [118, 119]. It runs on a cluster of PCs. The algorithms are described using web terminology. However, the technique is applicable in a larger setting to any graph. Furthermore, we believe that versions of the on-line algorithm running in a distributed environment could be useful in network applications when a link matrix is distributed between various sites.

Now consider the issue of archiving the web. In Section 7.6.3, we mention studies that we conducted with librarians from the French national Library to decide if page importance (PageRank) can be used to detect web sites that should be archived. In Chapter 8, we discuss other criteria of importance, such as site-based importance.

The chapter is organized as follows. We first present the model and in particular, recall the definition of importance. In Section 7.3, we introduce the algorithm focusing on static graphs. In Section 7.4, we consider different crawling strategies. In Section 7.5, we move to dynamic graphs, i.e., graphs that are continuously updated like the web. The following section deals with implementation and discusses some experiments. The last section is a conclusion.

7.2 Model

In this section, we present the formal model. Reading this section is not mandatory for the comprehension of the rest of the chapter.

The web as a graph We view the World Wide Web as a directed graph G. The web pages are the vertices. A link from one page to another form a directed edge.

We say that a directed graph G is *connected* if, when directed edges are transformed into non-directed edges, the resulting graph is connected in the usual sense. A directed graph G is said to be *strongly connected* if for all pair of vertices i, j there exists a directed path going from i to j following the directed edges of G. A graph is said to be *aperiodic* if there exists a k such that for all pair of vertices i, j there exists a directed path of length exactly k going from i to j following the directed edges of G. Thus aperiodicity implies strong connectivity.

When the web graph is not connected, each connected components may be considered separately.

A graph as a matrix Let G be any directed graph with n vertices. Fix an arbitrary ordering between the vertices. G can be represented as a matrix $L[1..n, 1..n]$ such that:

- L is nonnegative, i.e. $\forall i, \forall j, L[i, j] \geq 0$

- $L[i, j] > 0$ if and only if there is an edge from vertex i to vertex j.

There are several *natural* ways to encode a graph as a matrix, depending on what property is needed afterwards. For instance, Google [16, 88] defines the out-degree $d[i]$ of a page as the number of outgoing links, and set $L[i, j] = 1/d[i]$ if there is a link from i to j. In [62], Kleinberg proposes to set $K[i, j] = 1$ if there is a link from i to j, but then sets $L = K^T * K$ (where K^T is the transpose of matrix K).

Importance The basic idea is to define the importance of a page in an inductive way and then compute it using a fixpoint. If the graph contains n nodes, the importance is represented as a vector $\bar{x}$ in a n dimensional space. We consider three examples, in which the importance is defined inductively by the equation $\bar{x}_{k+1} = L\bar{x}_k$:

- If one decides that a page is important if it is pointed by important pages. Then set $L[i, j] = 1$ iff there is an edge between i and j.
- A "random walk" means that we browse the web by following one link at a time, and all outgoing links of a page have equal probability to be chosen. If one decides that a page importance is the probability to read it during a "random walk" on the web, then set $L[i, j] = 1/d[i]$ iff there is a edge between i and j. The random walk probabilities correspond to the Markov chain with generator L. This definition of L will result in the definition of importance as in Google PageRank.
- If one decides that a page is important if it is pointed by important pages or points to important pages. Then set $L[i, j] = 1$ iff there is an edge between i and j or an edge between j and i. This is related to the work of Kleinberg.

In all cases, this leads to solving by induction an equation of the type $\bar{x} = L\bar{x}$ where L is a nonnegative matrix. This suggests iterating over x_k, $\bar{x}_{k+1} = L\bar{x}_k$. Unfortunately, for obvious modulus reasons, this is very likely to diverge or to converge to zero. Observe that we are only interested in the relative importance of pages, not their absolute importance. This means that only the direction of $\bar{x}_k$ is relevant, not its norm. Thus it is more reasonable to consider the following induction (equivalent for importance computation), which uses the previous induction step but renormalizes after each step:

$$\bar{x}_{k+1} = \frac{L\bar{x}_k}{\| L\bar{x}_k \|} \quad (\dagger)$$

Computing the importance of the pages thus corresponds to finding a fixpoint $\bar{x}$ to $(\dagger)$, each i^{th} coordinate of x being the importance of page i. By definition, such a fixpoint is an eigenvector of L with a real positive eigenvalue. If $\bar{x}_0$ is a linear combination of all eigenvector having a real positive eigenvalue then it is easy to see that $(\dagger)$ will converge to the eigenspace corresponding to the *dominant* eigenvalue (i.e. which is maximal). Thus, unless x_0 is not general enough (e.g. not zero), the importance corresponds to an eigenvector of L which eigenvalue is a positive real and which modulus is maximal among all other eigenvalues.

For each nonnegative matrix L, there always exists such an eigenvector (see Perron-Frobenius theorem 7.2.1) but several problems may occur:

- There might be several solutions. This happens when the vector space corresponding to the maximal eigenvalue has a dimension greater than 1.
- Even if there is a unique solution, the iteration $(\dagger)$ may not converge when the graph does not have some desired properties.

All these cases are completely characterized in the Theorem of Perron-Frobenius that we give next.

Theorem 7.2.1
Perron-Frobenius [48]. Let L be an nonnegative matrix corresponding to a graph G. There exists an eigenvalue r which is real positive and which is greater than the modulus of any other eigenvalue. Furthermore,

1. *If G is strongly connected then the vector space for r is of dimension 1.*
2. *If G is aperiodic and $\bar{x}_0$ general enough (e.g. not zero) then the induction $(\dagger)$ converges towards **the** eigenvector for r of modulus 1. Note that the converse is true in the sense that if the graph is not aperiodic it is always possible to find an x_0 such that $(\dagger)$ does not converge.*

In order to solve the convergence problem, Google [49] uses the following patch. Recall that L is defined in this case by $L[i, j] = 1/d[i]$ iff there is an edge from i to j. A new matrix L' is defined such that $L'[i, j] = L[i, j] + \epsilon$

where ϵ is a small real. Then the fixpoint is computed over L' instead of L. Note that L' corresponds to a new graph G' which is G plus a "small" edge for any pair i, j. Observe that the new graph G' is strongly connected and aperiodic thus the convergence of $(\dagger)$ is guaranteed by Theorem 7.2.1. For each ϵ, this gives an importance vector $\bar{x}_\epsilon$. It is not hard to prove that when ϵ goes to zero, $\bar{x}_\epsilon$ converges to an eigenvector of L with a maximal real positive value. Thus, for ϵ small enough, $\bar{x}_\epsilon$ may be seen as a good approximation of the importance. For some mysterious reason, Google sets[2] ϵ to 0.2.

Another way to cope with the problem of convergence is to consider the following convergence suite:

$$(\dagger') \quad \bar{y}_{n+1} = \frac{Ly_n + y_n}{\parallel Ly_n + y_n \parallel}$$

If r is the maximal eigenvalue of a nonnegative matrix L then $r + 1$ can be shown to be the maximal eigenvalue of $L + I$. Thus, a solution $\bar{y}$ of $(\dagger')$ is also a solution of $\dagger$. If L is strongly connected then $L + I$ is aperiodic and thus $(\dagger')$ converges towards the importance. If L is not strongly connected there might be several linearly independent eigenvector, but still it is easy to show that $(\dagger')$ converges towards the projection of $\bar{x}_0$ on the eigenspace corresponding to all solutions.

On the web The computation of page importance in a huge dynamic graph has recently attracted a lot of attention because of the web, e.g., [80, 16, 88, 20, 41]. It is a major issue in practice that the web is *not* strongly connected. For instance, in the bow tie [17] vision of the web, the *OUT* nodes do not branch back to the *core* of the web. Although the same computation makes sense, it would yield a notion of importance without the desired semantics. Intuitively, the random walk will take us out of the core and would be "trapped" in *OUT* pages that do not lead back to the core (the "rank sink" according to [16]). So, pages in the core (e.g., the White House homepage) would have a null importance. Hence, enforcing strong connectivity of the graph (by "patches") is more important from a semantic point of view than for mathematical reasons. In a similar way to Google, our algorithm enforces the strong connectivity of the graph by introducing "small" edges. More precisely, in our graph, each node points to a unique virtual page. Conversely, this virtual page points to all other nodes.

[2]Greater values of ϵ increase the convergence speed.

Our Algorithm Our algorithm computes the characteristic vector of $(\dagger')$, and doesn't require any assumption on the graph. In particular, it works for any link matrix L, assuming that L can be read line by line. More precisely, for each page i that is read, we use the values $L[i,j]$ where $L[i,j] > 0$. For instance, in Google's link matrix, these values correspond to outgoing links (the pages j pointed by page i), which are known at little cost by parsing the HTML file. However, the cost may be higher in some other cases (e.g., when $L[i,j] > 0$ represents incoming links, we need to store and read an index of links). In terms of convergences, the different cases are characterized in a similar way as previously, e.g. if G is strongly connected, the solution is unique and independent of the initial vector x_0.

Previous work is abundant in the area of Markov chains and matrix fixpoint computations, e.g. [32] or [80]. In most cases, infinite transition matrix are managed by increasing the size of a known matrix block. Some works also consider a changing web graph, e.g. an incremental computation of approximations of page importance is proposed in [30]. As far as we know, our algorithm is new. In particular:

- it may start even when a (large) part of the matrix is still unknown,
- it helps deciding which (new) part of the matrix should be acquired (or updated),
- it is integrated in the crawling process,
- it works on-line even while the graph is being updated.

For instance, after crawling 400 million pages on the web, we have a relatively precise approximation of page importance for over 1 billion pages, i.e., even of parts of the matrix corresponding to pages that we did not read so far.

7.3 Static graphs: OPIC

We consider in this section the case of a static graph (no update). We describe the algorithm for Google link matrix L as defined previously. It can be generalized to work for other link matrices. We present the OPIC algorithm and show its correctness. OPIC stands for Online Page Importance Computation. We briefly discuss the advantages of the technique over the off-line algorithm. We will consider dynamic graphs in the next section.

Informal description

For each page (each node in the graph), we keep two values. We call the first *cash*. Initially, we distribute some cash to each node, e.g., if there are n nodes, we distribute $1/n$ to each node. While the algorithm runs, the cash of a node records the recent information discovered about the page, more precisely, the sum of the cash obtained by the page since the last time it was crawled. We also record the *(credit) history* of the page, the sum of the cash obtained by the page since the start of the algorithm until the last time it was crawled. The cash is typically stored in main memory whereas the history may be stored on disk. When a page i is retrieved by the web agent, we know the pages it points to. In other words, we have at no cost the outgoing links information for the retrieved page. We record its cash in the history, i.e., we add it to the history. We also distribute this cash equally between all pages it points to. We reset the cash of the page i to 0. This happens each time we read a page. We will see that this provides enough information to compute the importance of the page as used in standard methods. We will consider in a further section how this may be adapted to handle dynamic graphs.

Detailed description

We use two vectors $C[1..n]$ (the cash) and $H[1..n]$ (the history). The initialization of C has no impact on the result. The history of a page is simply a number. A more detailed history will be needed when we move to an adaptive version of the algorithm. Let us assume that the history H is stored on disk and C is kept in main memory. In order to optimize the computation of $|H| = \sum_i H[i]$, a variable Z is introduced so that $Z = |H|$ at each step. The algorithm is shown in Figure 7.1.

In this algorithm, we use $|H| = \sum_i H[i] = Z$. At each step, an estimate of any page k's importance is $(H[k] + C[k])/(Z + 1)$.

Note that the algorithm does not impose any requirement on the order we visit the nodes of the graph as long as each node is visited infinitely often (some minimal *fairness*). This is essential since crawling policies are often governed by considerations such as robots exclusion, politeness (avoid rapid-firing), page change rate, focused crawling.

As long as the cash of children is stored in main memory, no disk access is necessary to update it. At the time we visit a node (we crawl it), the list of its

```
for each i let C[i] := 1/n ;
for each i let H[i] := 0 ;
let Z:=0 ;

do forever
begin
    choose some node i ;
    %% each node is selected
    %% infinitely often

    H[i] += C[i];
    %% unique read/write disk ac-
cess per page

    for each child j of i,
        do C[j] += C[i]/out[i];
    %% Distribution of cash
    %% depends on L

    Z += C[i];
    C[i] := 0 ;
end
```

Figure 7.1: On-line Page Importance Computation

children is available on the document itself and does not require disk access. Two disk accesses (one read, one write) are needed for updating the history of the page that is visited. However, note that our crawler needs anyway to read and write some metadata (e.g. the date of crawl) for each page that is visited. Thus, given that the history value is stored with the other metadata, updating it does not add any cost in terms of disk access.

Each page has at least one child, thanks to the “small” edges that we presented in the previous Section (and that points to the virtual page). However, for practical reasons, the cash of the virtual page is not distributed all at once. This issue is in particular related to the discovery of new pages and management of variable sized graphs that we consider later.

We next prove the correctness of the algorithm.

Proof of correctness

Consider a graph G of n nodes. We will use the following notation:

Notation 7.3.1
We note C_t and H_t the values of vectors C and H at the end of the t-th step of the algorithm. The vector C_0 denotes the value of vector C at initialization (all entries are $1/n$). Let X_t be defined by:

$$X_t = \frac{H_t}{|H_t|}, \text{ i.e., } \forall j, X_t[j] = \frac{H_t[j]}{(\sum_i H_t[i])}$$

Assuming the graph is strongly connected, when t goes to infinity, we will see that:

- $|H_t|$ goes to infinity
- $$|(L' * X_t) - X_t| < \frac{1}{|H_t|}$$
- $|X_t| = 1$.

This will show the following theorem:

Theorem 7.3.1
The vector X_t converges to the vector of importance, i.e.,

$$\exists Imp, lim_{t \to +\infty} X_t = Imp$$

Recall that the algorithm assumes that all pages are read infinitely often. In other words, for each time t, for each page k, there exists some time $t' > t$ such that page k will be read at time t'.

$$\forall t, \forall k, \exists t', t' > t, choosenode(t') = k$$

To prove this theorem, we use the five following lemmas:

Lemma 7.3.2
The total amount of all cash is constant and equal to the initial value, i.e., for each t, $\sum_{i=1}^{n} C_t[i] = \sum_{i=1}^{n} C_0[i] = 1$

Proof: This is obvious by induction since we only distribute each node cash among the children.

Lemma 7.3.3
After each step t, we have for each page j,

$$H_t[j] + C_t[j] = C_0[j] + \sum_{(i\ ancestor\ of\ j)} \left(\frac{L[i,j]}{out[i]} * H_t[i]\right)$$

Proof: The proof is by induction. Clearly, the lemma is true at time $t = 0$. Suppose it is true at time t for each element j. At step $t+1$, some page k is crawled. We prove the formula holds at time $t+1$ for each element j. We consider the two cases: j equals k or not.

$j = k$ If $j = k$, then the right term doesn't change: $\forall i, i \neq j, H_{t+1}[i] = H_t[i]$. The left term value doesn't change either, the cash is added to H and then set to zero. So $H_{t+1}[j] + C_{t+1}[j] = H_t[j] + C_t[j]$, and the equation is true at $t+1$.

$j \neq k$ Then $C_{t+1}[j]$ increases by $C_t[k] * \frac{L[i,j]}{out[i]}$. So

$$H_{t+1}[j] + C_{t+1}[j] = C_0[j] + \sum_{(i\ ancestor\ of\ j)} \left(\frac{L[i,j]}{out[i]} * H_t[i]\right) + C_t[k] * \frac{L[k,j]}{out[k]}$$

Now $\forall i, i \neq k, H_{t+1}[i] = H_t[i]$, and also $H_{t+1}[k] = H_t[k] + C_t[k]$, and this shows the result.

Lemma 7.3.4
If all pages are infinitely read, $\sum_j H_t[j]$ goes to infinity.

Proof: This lemma is true if there exists $e > 0$ such that starting at any time t, $\sum_j H_t[j]$ will eventually increase of e. Consider $e = 1/n$, i.e. e is the average value of cash on all pages. Let t be any time. At time t, there is a page j having more than e cash. By definition of the algorithm, the cash of page j can not decrease until j is read. Moreover, the page j will be read one more time after t because all pages are read infinitely often. Thus, the history of the page will increase of at least e when page j is read, and this will increase $\sum_j H_t[j]$.

Note that for Lemma 7.3.4, it is not necessary that G is a strongly connected graph: the proof works for any graph. Considering a strongly connected graph, a stronger result is obtained as follows.

Lemma 7.3.5
Consider a strongly connected graph G with n nodes.

(a) Let i, j be any pair of nodes. Then, c in the cash of node i eventually leads to c/n^n in the cash of node j.

(b) As a consequence, for each node j, $H_t[j]$ goes to infinity.

Proof: In the algorithm, each node of the graph, when it is read, splits the value by at most n, because it can't have more than n different links. We suppose that the graph is strongly connected, so there is a path from i to j, and it is no longer than n. Let's note $P_1 ... P_k$ the pages for this path. Thus, each time page P_x is crawled with some cash c, the cash of page P_{x+1} is increased by at least c/n. Now we suppose that every page is crawled an infinite number of times. Consider some time t. We eventually we will crawl P_1 at time $t_1 > t$, then eventually P_2 at time $t_2 > t_1$, ... until P_k. Thus we will eventually have distributed at least c/n^n in the cash of j. This shows (a).

Consider any moment t, some node contains at least $1/n$ cash (because $\sum_i C_t[i] = 1$). Thus, it will eventually increase the cash of j (thus eventually its history) by $1/n^n$. In other words, there is a positive constant e (for instance $e = 1/n^n$) such that for each page j, for each time t, we will eventually increase the history of j by e at some time $t' > t$. Thus $H_t[j]$ goes to infinity. This shows (b). This is stronger than Lemma 7.3.4 in that we show here that $H[j]$ goes to infinity for each node j, whereas Lemma 7.3.4, only $|H|$ goes to infinity.

Now it is possible to describe the limit when t goes to infinity.

Lemma 7.3.6
$\lim_{t \to +\infty} |L' * X_t - X_t| = 0$

Proof: By definition of X_t, for each i, $X_t[i] = H_t[i] / \sum H_t[j]$. Then,

By Lemma 7.3.3,

$$H_t[j] + C_t[j] = C_0[j] + \sum_{(i \ ancestor \ of \ j)} \left(\frac{L[i,j]}{out[i]} * H_t[i]\right)$$

Let us look at the jth coordinate of $|L' * X_t - X_t|$:

$$\left|\frac{(L' * H_t - H_t)[j]}{\sum_k H_t[k]}\right| = \left|\frac{C_t[j] - C_0[j]}{\sum_k H_t[k]}\right| \leq \frac{1}{\sum_k H_t[k]}$$

Its limit is 0 because, when t goes to infinity, $\sum_j H_t[j]$ goes to infinity (by Lemma 7.3.4) and $C_0[j]$, $C_t[j]$ are bounded by 1.

By Lemma 7.3.6, X_t goes infinitely close to a characteristic vector of L of the dominant characteristic value r. This suggests using $X_t = H_t/Z$ as an estimate of page importance.

Theorem 7.3.7
The limit of X_t is Imp, i.e., $lim_{t\to+\infty} X_t = Imp$

Proof: By the previous result,

$$\lim_{t\to+\infty} |(L' - 1) * X_t| = 0$$

where 1 is the identity matrix (1 in the diagonal and 0 elsewhere). Consider now the decomposition of $X_t = S_t + D_t$ where S_t is in $Ker(L' - 1)$ (the kernel of matrix $L' - 1$), and D_t in the corresponding orthogonal space where the restriction of $L' - 1$ is invertible. Because S_t is in $Ker(L' - 1)$, we have $\forall t, L' * X_t - X_t = L' * D_t - D_t$ and so

$$\lim_{t\to+\infty} |(L' - 1) * D_t| = 0$$

We can now restrict to the orthogonal space of $Ker(L' - 1)$, in which $L' - 1$ has an inverse called H. The matrix multiplication being continuous, we can multiply to the left by H, which is constant, and thus

$$\lim_{t\to+\infty} |D_t| = 0$$

Now if we use the fact that there is a single fixpoint solution for L', that mean that $Ker(L' - 1)$ is of dimension 1 and that

$$\forall t, X_t = \alpha_t * Imp + D_t$$

where α_t is a scalar. Now because $|D_t|$ converges to zero, and $|X_t| = |Imp| = 1$, we have:

$$\lim_{t\to+\infty} X_t = Imp$$

Note that we can add 1 (i.e. $\sum_i C_t[i]$) to the denominator Z by using the cash accumulated since last crawl, and thus have (on average) a marginally better estimate. More precisely, one can use for page j,

$$\frac{H_t[j] + C_t[j]}{(\sum_i H_t[i]) + 1}$$

We will mention some details of the implementation in Section 7.6. We can already mention advantages over the off-line algorithm. OPIC uses only local information, i.e. the outgoing links of the page that is being crawled that can be found in the page as URLs. Thus our algorithm presents the following advantages:

Advantages over the off-line algorithms The main advantage of our algorithm is that it allows focused crawling. Because our algorithm is run online and its results are immediately available to the crawler, we use it to focus crawling to the most interesting pages for the users. This is in particular interesting in the context of building a web archive [4], when there are strong requirements (and constraints) on the crawling process.

Moreover, since we don't have to store the matrix but only a vector, our algorithm presents the following advantages:

1. It requires less storage resources than standard algorithms.
2. It requires less CPU, memory and disk access than standard algorithms.
3. It is easy to implement.

Our algorithm is also well adapted to "continuous" crawl strategies. The reason is that storing and maintaining the link matrix during a "continuous" crawl of the web (when pages are refreshed often) is significantly more expensive than for single "snapshot" crawl of the web (when each page is read only once). Indeed, when information about specific pages has to be read and updated frequently, the number of random disk access may become a limiting factor. In our experiment for instance, the crawler was retrieving hundreds of pages per seconds on each PC (see Section 7.6). However, note that the storage of a link matrix may be useful beyond the computation of page importance. For instance, given a page p, Google provides the list of pages pointing to it. This means that the matrix (or its transpose) is maintained in some form. Another usage of the link matrix is exhibited in [42].

7.4 Crawling Strategies

In this section, we first consider different crawling strategies that impact the speed of convergence of our algorithm. Then, we study how they can be used in the case of a changing graph. Implementations aspects and experiments are considered in the next section.

7.4.1 On convergence

As previously mentioned, the error in our estimate is bounded by $\frac{1}{|H_t|}$. Let us call $\frac{1}{Z_t} = \frac{1}{|H_t|} = 1/\sum_k H_t[k]$ the *error factor*, although this is, strictly speaking, not the error (but an upper bound for it). Now, in principle, one could choose a very bad strategy that would very often select pages with very low cash. (The correctness of the algorithm requires that each page is read infinitely many times but does not require the page selection strategy to be smart.) On the other hand, if we choose nodes with very large cash, the error factor decreases faster.

To illustrate, consider three page selection strategies:

1. *Random* : We choose the next page to crawl randomly with equal probability. (Fairness: for each t_0, the probability that a page will be read at some $t > t_0$ goes to 1 when t goes to infinity.)

2. *Greedy* : We read next the page with highest cash. This is a greedy way to decrease the value of the error factor. (Fairness: For a strongly connected graph, each page is read infinitely often because it accumulates cash until it is eventually read. See Lemma 7.3.5 in the appendix).

3. *Cycle* : We choose some fixed order and use it to cycle around the set of pages. (Fairness is obvious.) We considered this page selection strategy simply to have a comparison with a systematic strategy. Recall that systematic page selection strategies impose undesired constraints on the crawling of pages.

Remark 7.4.1

(Xyleme strategy) The strategy for selecting the next page to read used in Xyleme is close to *Greedy*. It is tailored to optimize our knowledge of the web [89], the interest of clients for some portions of the web, and the refreshing of the most important pages that change often.

Random vs. Greedy. To get a feeling of how *Random* and *Greedy* progress, let us consider some estimates of the values of the error factor for these two page selection strategies. Suppose that at initialization, the total value of the cash of all pages is 1 and that there are n pages. Then:

- *Random* : The next page to crawl is chosen randomly so its cash is on average $\frac{1}{n}$. Thus, the denominator of the error factor is increased by $\frac{1}{n}$ on average per page.
- *Greedy* : A page accumulates cash until it reaches the point where it is read. Let α be the average cash of a page at the time it is read. On average, the cash of the page is $\alpha/2$ if we suppose that cash is accumulated linearly by pages until they are read. This result has been confirmed by experiments. Since the total of the cash is 1, this shows that α is $2 * (1/n)$. Thus the denominator of the error factor is increased by $\frac{2}{n}$ on average per page read. This result has also been confirmed by experiments, the average "cash" value of pages at the time they are crawled is close to $\frac{2}{n}$.

Thus the error factor decreases on average twice faster with *Greedy* than with *Random*. We will see with experiments (in Section 7.6) that, indeed, *Greedy* converges faster. Moreover, *Greedy* focuses resources on important pages. For these pages, it outperforms *Random* even more.

7.5 A changing graph: The Adaptive OPIC algorithm

Consider now a dynamic graph (the case of the web). Pages come and disappear and edges too. Because of the time it takes to crawl the web (weeks or months), our knowledge of the graph is not perfect. Page importance is now a moving target and we only hope to stay close to it.

It is convenient to think of the variable $Z = |H|$ as the clock. Consider two time instants $t-T, t$ corresponding to Z having the value $t-T$ and t. Let $H_{t-T,t}[i]$ be the total of cash added to the history of page i between time $t-T$ and t, i.e., $H_t[i] - H_{t-T}[i]$. Let

$$\forall j, X_{t,T}[j] = \frac{H_{t-T,t}[j]}{(\sum_i H_{t-T,t}[i])} = \frac{H_{t-T,t}[j]}{T}$$

Because the statement of Theorem 7.3.3 does not impose any condition on the initial state of X_t, it is obvious that $X_{t,T}$ converges to the vector of importance when T goes to infinity. (Note that on the other hand, for a fixed T, when t goes to infinity, $X_{t,T}$ does not converge to the vector of importance.) Using the data gathered between $t-T$ and t, comes to ignoring the history before time $t-T$ and starting with the state of the cash at time $t-T$ for initial state. Observe that this state may be not more informative than the very first state with equal distribution of cash.

We thus estimate the importance of a page by looking at the history between t (now) and $t-T$. We call the interval $[t-T, t]$ the (time) *window*. There is a trade-off between precision and adaptability to changes and a critical parameter of the technique is the choice of the size of the window.

The Adaptive OPIC algorithm We next describe (variants of) an algorithm, namely Adaptive OPIC, that compute(s) page importance based on a time window.

In Adaptive OPIC, we have to keep some information about the history in a particular time window. We considered the following window policies:

- Fixed Window (of size T): For every page i, we store the value of cash $C_t[i]$ and the global value Z_t for all times it was crawled since (*now* - T).
- Variable Window (of size k): For every page i, we store the value of cash $C_t[i]$ and the global value Z_t for the last k times this page was crawled.
- Interpolation (of time T): For every page i, we store only the Z_t value when it was last crawled, and an interpolated history $H[i]$ (to be defined) that estimates the cash it got in a time interval of size T before that last crawl.

In the following, we call *measure* a pair (C, Z). Note that in Variable Window, we store exactly k measures; and that in Interpolation, we store only one. Note also that in Fixed Window, the number of measures varies from one page to another, so this strategy is more complex to implement.

In our analysis of Adaptive OPIC, there will be two main dimensions: (i) the page selection strategy that is used (e.g., *Greedy* or *Random*) and (ii) the window policy that is considered (e.g., Fixed Window or Interpolation).

Fixed Window One must be aware that some pages will be read rarely (e.g., once in several months), whereas others will be read perhaps daily. So there are

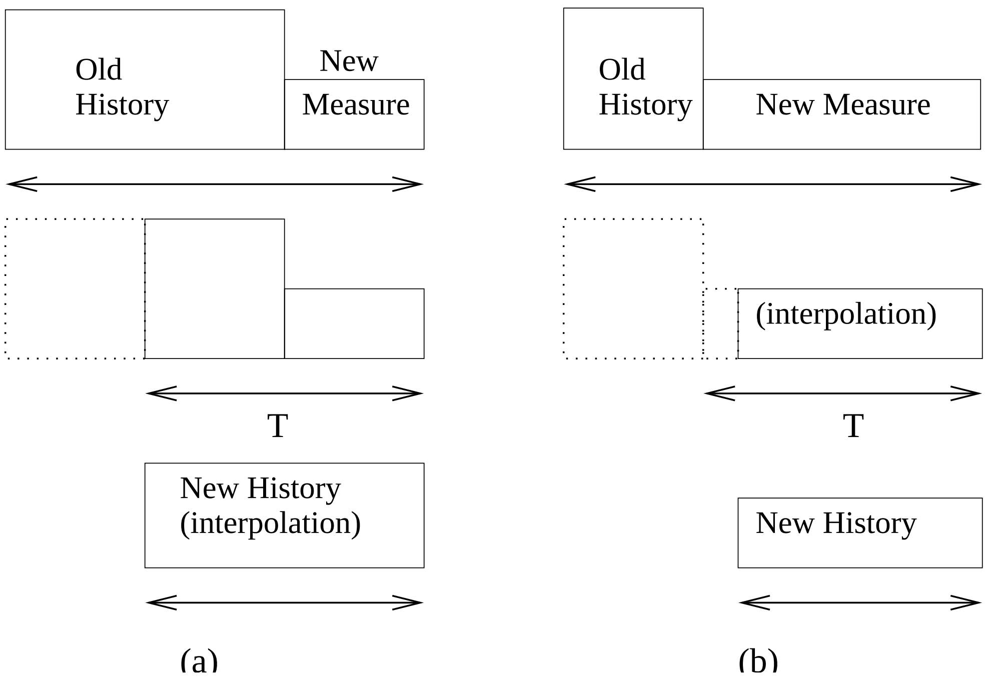

Figure 7.2: Simple Interpolation

huge variations in the size of histories. For very large histories, it is interesting to use compression techniques, e.g., to group several consecutive measures into one. On the opposite, we have too few measures for very unimportant pages. This has a negative impact on the speed of convergence of the algorithm. By setting a minimum number of measures per page (say 3), experiments show that we obtain better results. See Section 7.6.

Interpolation It is tailored to use little resources. Indeed, for each page, the history simply consists of two values. This is what we tested on real web data (See Section 7.6). It is the policy actually used in Xyleme [118, 89, 119]. It is based on a fixed time window of size T. The algorithm uses for history two vectors $H[1..n], Z[1..n]$:

- $H[i]$ represents the sum of the cash acquired by the page i during a time period T before the last crawl. This value is obtained by interpolation.
- $Z[i]$ is the Z-time of that last crawl.

When we visit a page and update its history, we *estimate* the cash that was added to that page in the interval T until that visit. See Figure 7.2 for an intuition of the interpolation. We know what was added to its cash between time $Z[i]$ and Z, $C[i]$. The interpolation assumes that the page accumulates cash linearly. This has been confirmed by experiments. More precisely, the history is updated as follows:

$$\begin{array}{ll} H[i] * \frac{T-(Z-Z[i])}{T} + C[i] & \text{if } Z - Z[i] < T \\ C[i] * \frac{T}{Z-Z[i]} & \text{otherwise} \end{array}$$

Expanding the graph When the number of nodes increases, the relative difficulty to assign a cash and a history to new nodes highlights some almost philosophical issues about the importance of pages. Consider the definition of importance based on $(\dagger)$. When we crawl new pages, these pages acquire some importance. The importance of previously known pages mechanically decreases in average simply because we crawled more pages. This is true for instance in the random walk model: adding new pages of non-null probability to be read can only decrease the probability of other pages to be read. However, these changes in pages importance seem unfair and are not expected by users of the system. We assign to each new page a default history that corresponds to the importance of recently introduced pages. Experiments confirmed this to be a good estimate. The reason is that important pages are discovered first, whereas new or recently introduced pages are often the least important ones.

Focused crawling and page discovery In our system, the scheduling of pages to be read depends mostly on the amount of "cash" for each page. The crawling speed gives the total number of pages that we can read for both discovery and refresh. Our page importance architecture allows us to allocate resources between discovery and refresh. For instance, when we want to do more discovery, we proceed as follows: (i) we take some cash from the virtual page and distribute it to pages that were not read yet (ii) we increase the importance of "small" edges pointing to the virtual page so that it accumulates more cash. To refresh more pages, we do the opposite. We can also use a similar method to focus the crawl on a subset of interesting pages on the web. For instance, we used this strategy to focus crawling to XML pages [118, 89]. In some other applications, we may prefer to quickly detect new pages. For instance, we provide to a press agency a "copy tracker" that helps detecting copies of their News wires over the web. The problem with News pages is that they often last only a few days. In this particular application, we process as follows for each link: pages that are suspected to contain news wires (e.g. because the URL contains “news”) receive some “extra” cash. This cash is taken from the (unique) virtual page so that the total value of cash in the system does not change.

7.6 Implementation and experiment

We implemented and tested first the standard off-line algorithm for computing page importance, then variants of Adaptive OPIC. We briefly describe some aspects of the implementation. We then report on experiments first on synthetic data, then on a large collection of web pages.

7.6.1 A distributed implementation

We implemented a distributed version of Adaptive OPIC that can be parameterized to choose a page selection strategy, a window policy, a window size, etc.

Adaptive OPIC runs on a cluster of Linux PCs. The code is in C++. Corba is used for communications between the PCs. Each crawler is in charge of a portion of the pages of the web. The choice of the next page to read by a crawler is performed by a separate module (the Page Scheduler). The split of pages between the various crawlers is made using a hash function h_{url} of the URLs. Each crawler evaluates the importance of pages it is in charge of. Its portion of the cash vector is in main memory, whereas its portion of the history is on disk. The crawler also uses an (in memory) hash table that allows to map a URL handled by this crawler to its identifier (an integer) in the system. Finally, it uses a map from identifiers to URLs. This last map may reside on disk. Each crawler crawls millions of pages per day. The bandwidth was clearly the limiting factor in the experiments. For each page that is crawled, the crawler receives the identifier of a page from the page scheduler and then does the following:

Fetch: It obtains the URL of the page, fetches the page from the web and parses it;

Money transfers: It distributes the current cash of the page to the pages it points to. For each such page, it uses h_{url} to obtain the name of the server in charge of that page. It sends a “money transfer” to that server indicating the URL of the page and the amount. This is a buffered network call.

Records: It updates metadata (e.g. date of crawl, hash signature) about the visited page. This requires a pair of disk access (a read and a write). The history of the page, stored with the other metadata, is also updated. The cash is reset to null.

Each crawler also processes the money transfer orders coming from other servers. Communications are asynchronous.

It should be observed that for each page crawled, there are only two disk accesses, one to obtain the metadata of the page and one to update the metadata, including the history. Besides that, there are Corba communications (on the local network), and main memory accesses.

7.6.2 Synthetic data

Although we started our experiments with a large collection of URLs on the web, synthetic data gave us more flexibility to study various input and output parameters, such as: graph size, graph connectivity, change rates, types of changes, distribution of in-degrees, out-degrees and page importance, importance error, ranking errors. So, we report on them first.

The graph model We performed experiments with various synthetic graphs containing dozens of millions of web pages. These experiments showed that the use of very large graphs did not substantially alter the results.

For instance, we started with graphs obtained using a Poisson distribution on the average of incoming links, a somewhat simplistic assumption. We then performed experiments with more complex distributions following recent studies of the web graph [17], e.g., with a power distribution $P(I = n) = 1/n^{2.1}$. Results were rather similar to those obtained using a Poisson distribution. In order to also control the distribution of outgoing links and the correlations between them, we tried several graph models in the spirit of [37], but even with significant changes of the graph parameters, the patterns of the results did not change substantially from the simple graph model. So, we then restricted our attention to rather simple graphs of reasonably small size to be able to test extensively, e.g., various page selection strategies, various window sizes, various patterns of changes of the web.

In the remaining of this section, we will consider a simple graph model based on the power distribution on incoming edges. Details omitted. The number of nodes is fixed to N = 100 000 nodes.

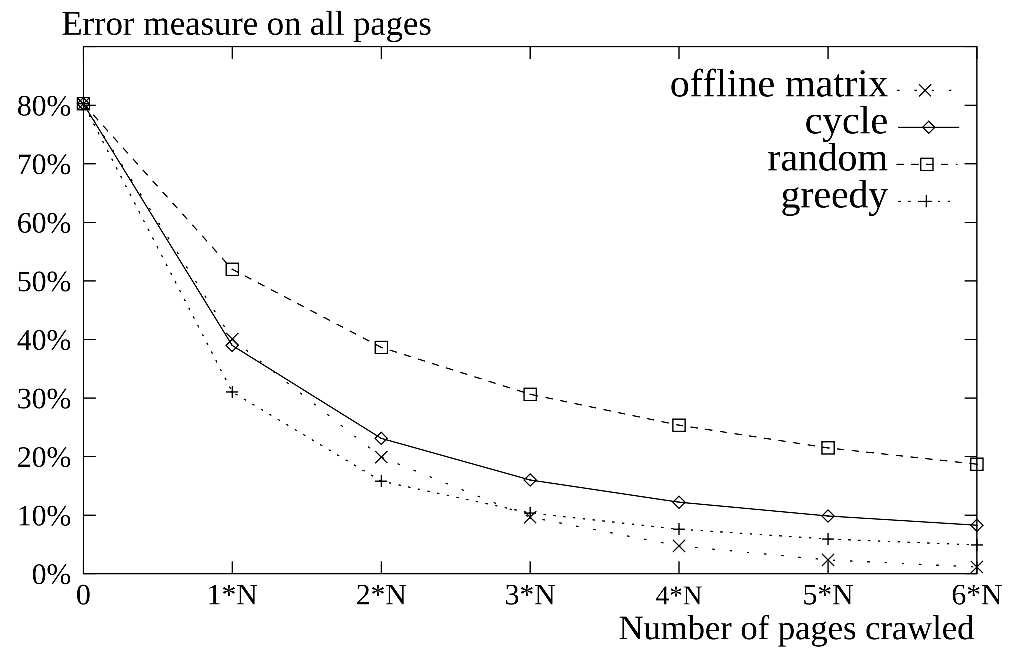

Figure 7.3: Convergence of OPIC (on all pages)

Impact of the page selection strategy First, we studied the convergence of OPIC for various page selection strategies. We considered *Random*, *Cycle* and *Greedy*. We compared the values of the estimates at different points in the crawl, after crawling N pages, up to to $10 * N$ pages.

The error we compute is the mean over the set of pages of the error between the computation of OPIC at this state and the value of the fixpoint. More precisely, we compute the average of the percentage of error:

$$100 * \frac{\sum_j \frac{|X[j]-Imp[j]|}{Imp[j]}}{N}$$

where Imp is obtained by running the off-line algorithm until a fixpoint is reached (with negligible error).

Consider Figure 7.3. The error is about the same for *Greedy* and *Cycle*. This result was expected since previous studies [59] show that given a standard cost model, uniform refresh strategies perform as good as focused refresh. As we also expected, *Random* performs significantly worse. We also compared these, somewhat artificially, to the off-line algorithm. In the off-line, each iteration of the matrix is a computation on N pages, so we count N “crawled pages” for each iteration. The off-line algorithm converges almost like *Cycle* and *Greedy*. This

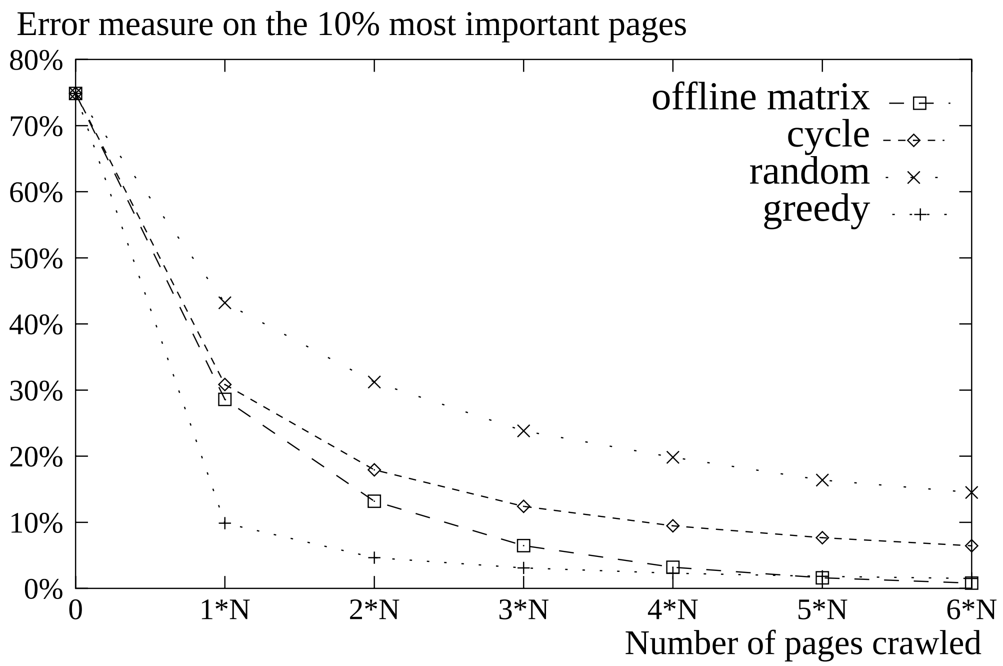

Figure 7.4: Convergence of OPIC (on important pages)

is not surprising since the crawl of N pages with *Cycle* corresponds roughly to a biased iteration on the matrix.

Now consider Figure 7.4. The error is measured now only for the top ten percent pages, the interesting ones in practice. For this set of pages, *Greedy* (that is tailored to important pages) converges faster than the others including the off-line algorithm.

We also studied the variance. It is roughly the same for all page selection strategies, e.g., almost no page had a relative error more than twice the mean error. We also considered alternative error measures. For instance, we considered an error weighted with page importance or the error on the relative importance that has been briefly mentioned. We also considered the error in ordering pages when their importance is used to rank query results. All these various error measures lead to no significant difference in the results.

Impact of the size of the window As already mentioned, a small window means more reactivity to changes but at the cost of some lack of precision. A series of experiments was conducted to determine how much. To analyze the impact of the size of the window, we use Adaptive OPIC with the *Greedy* strategy and a Fixed Window of M crawls, i.e., we keep for each page the history since the last

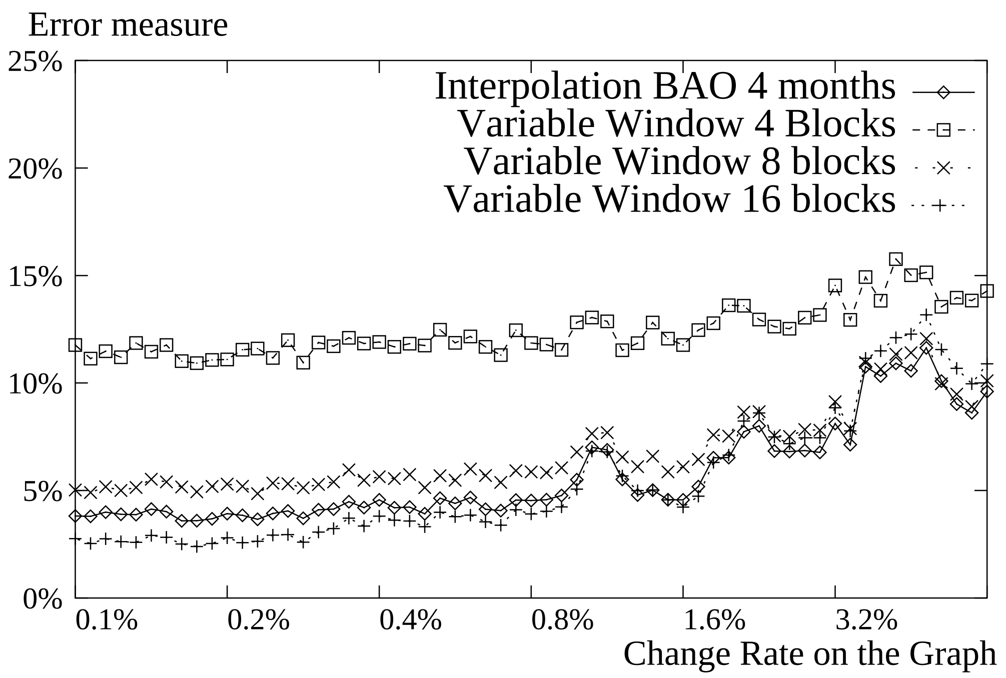

Figure 7.5: Influence of window's sizes

M crawls of the page. Similar results were obtained with other variants of the algorithm. Consider Figure 7.5 ignoring the Interpolation policy for the moment. The change rate is the number of pages that have their in-degree significantly modified (i.e. divided par two or multiplied by two) during the time of crawling N pages, where N is the number of pages on the graph (i.e. the time for "one" crawl of the graph). For each change rate the graph is crawled ten times. The figure shows the result for M = 4, 8, 16. The important point to notice is that we can get reasonably close to the fixpoint with rather small windows (e.g., $M = 8$ here). As previously mentioned, the trade-off is reactivity to changes versus precision. When the time window becomes too small (e.g., $M = 4$ here), the error is more important. This is because each measure for a page gives only a too rough estimate of this page importance, so the error is too large. Such an error may still be acceptable for some applications.

Now observe the Interpolation experiment in Figure 7.5. First, note that it performs almost as well as large Variable Window (e.g. $M = 16$) on graph with few changes. Also, it adapts better to higher change rates (e.g. more than 1 percent). So, let us consider now the comparison of various window policies.

Window Type and Size	**Measures per page**
Variable Window 8 measures	8
Fixed Window 8 months	8.4
Improved Fixed Window 4 months	6.1
Interpolation 4 months	1

Figure 7.6: Storage resources per time window

Impact of the window policy We compared different policies for keeping the history. In this report, we use again the *Greedy* strategy. Various window policies may require different resources. To be fair, we chose policies that roughly requested similar amount of resources. Typically, we count for storage the number of measures we store. (Recall that a measure consists of a value for C and one for Z.) The five policies we compared used between 4 and 8 measures, except Interpolation that by definition uses only 1. Figure 7.6 shows the average number of measures used per page in each case. These measures depend for Fixed Window on the crawling speed which was set here to be N pages per month (the speed was chosen here so that Fixed Window would use about as much resources as the others). We also considered a variant of Fixed Window that forces each page to have a minimum number of measures, namely Improved Fixed Window. We required for the experiment mentioned here a minimum of 3 measures. Note that this resulted for this particular data set in an increase of the average number of measures from 4 to 6.1.

Now consider Figure 7.7. It shows that for a similar number of measures, Variable Window performs better than Fixed Window. The problem with Fixed Window is that very few measures are stored for unimportant pages and the convergence is very slow because of errors on such pages. On the other hand, the Improved Fixed Window policy yields significantly better results. The improvement comes indeed from more reliability for unimportant pages.

The most noticeable result about the use of windows is that the algorithm with the Interpolation policy outperforms the other variants while consuming less resources. Indeed, the error introduced by the interpolation is negligible. Furthermore, the interpolation seems to avoid some “noise” introduced when an old measure is added (or removed) in Adaptive OPIC. In some sense, the interpolation acts as a filter on the sequence of measures.

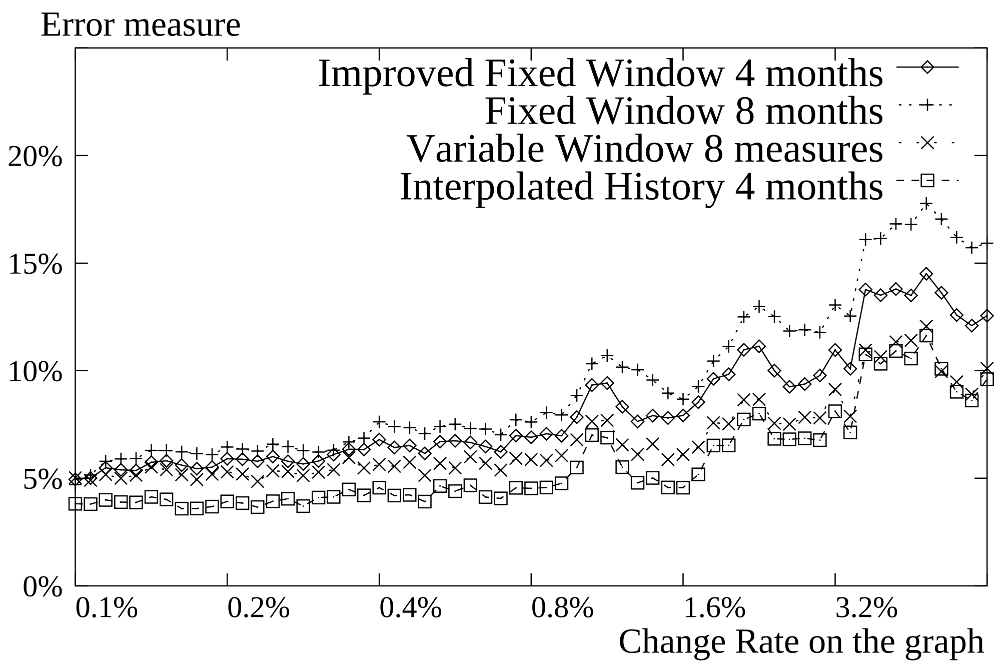

Figure 7.7: Influence of window's types

Of course the convergence of all variants of the adaptive algorithms depends on the time window that is used. The excellent behavior of Interpolation convinced us to adopt it for our experiments with crawls of the web. This is considered next.

7.6.3 Web data

We performed the web experiments using the crawlers of Xyleme [119]. The crawl used the page selection strategy of Xyleme that has been previously mentioned and is related to *Greedy*. The history was managed using the Interpolation policy.

During the test, the number of PCs varied from 2 to 8. Each PC had little disk space and less than 1.5Gb of main memory. Some reasonable estimate of page importance for the most important pages was obtained in a few days, as important pages are read more frequently and discovered sooner than others. The experiments lasted for several months. We discovered one billion URLs; only 400 millions of them were actually read. Note that because of the way we discover pages, these are 400 million relatively important pages. Moreover, we could give reasonable importance estimates even on pages that were never read. This experiment was sufficient (with limited human checking of the results) to conclude that the algorithm could be used in a production environment. Typically, for all practical uses of importance we considered (such as ranking query results or scheduling page refresh), the precision brought by the algorithm is rapidly sufficient. An advantage of the algorithm is also that it rapidly detects the new important pages, so they can be read sooner.

A main issue was the selection of the size of the time window. We first fixed it too small which resulted in undesired variations in the importance of some pages. We then used a too large window and the reactivity to changes was too limited. Finally, the window was set to 3 months. This value depends on the crawling speed, which in our case was limited by the network bandwidth.

Our performance analysis also showed that using our system (Xyleme crawler and Adaptive OPIC), it is possible to, for instance, crawl and compute page importance (as well as maintain this knowledge) for a graph of up to 2 billions pages with only 4 PCs equipped each with 4Gb of main memory and a small disk.

In the context of web archiving [4], we also conducted experiments to decide if our measures of page importance could be used to select pages of interest for the French national Library [73]. We selected thousand web sites, and 8 different professional librarians ranked each site in order to decide which sites should be archived (on a 1 to 4 scale). We defined the reference value for each site based on the average of these rankings. Finally, we defined the "score" of a librarian as the number of sites in which his rank was identical to the reference. The scores of librarians ranged from 60 to 80 percent, and the score of our page importance measures was 65 percent. This means that our measure based only on page importance was as good as a professional librarian, although not as good as the best ones. We are currently working on using other criteria [4] to improve the "automatic" librarian.

Other Improvements During our experiments, we found out that the semantics of links in dynamic pages is (often) not as good as in pages fully written by humans. Links written by humans usually points to more relevant pages. On the other hand, most links in dynamic pages often consist in other (similar) queries to the same database. For instance, forum archives or catalog pages often contain many links that are used to browse through classification. Similarly, we found out that "internal" links (links that point to a page on the same web site) are often less useful to discover other relevant pages than "external" links (links to a page on some other web site). Motivated by that, we proposed in [4] a notion of site-based importance that we describe in next Chapter. The main idea is to consider links between web-sites instead of links between web-pages. We are currently experimenting our algorithm with this new notion of importance per site.

7.7 Conclusion

We proposed a simple algorithm to implement with limited resources a realistic computation of page importance over a graph as large as the web. We demonstrated both the correctness and usability of the technique. Our algorithm can be used to improve the efficiency of crawling systems since it allows to focus on-line the resources to important pages. It can also be biased to take into account specific fields of interest for the users [4].

More experiments on real data are clearly needed. It would be in particular interesting to test the variants of Adaptive OPIC with web data. However, such tests are quite expensive.

To understand more deeply the algorithms, more experiments are being conducted with synthetic data. We are experimenting with various variants of Adaptive OPIC. We believe that better importance estimates can be obtained and are working on that. One issue is the tuning of the algorithms and in particular, the choice of (adaptable) time windows. We are also continuing our experiments on changing graphs and in particular on the estimate of the derivative of the importance. We finally want to analyze more in-depth the impact of various specific graph patterns as done in [75] for the off-line algorithm.

We are also working on a precise mathematical analysis of the convergence speed of the various algorithms. The hope is that this analysis will provide us with bounds of the error of the importance, and will also guide us in fixing the size of windows and evaluating the changes in importance. We are also improving the management of newly discovered pages.

The algorithm presented here computes page importance that depends on the entire graph by looking at one page at a time independently of the order of visiting the pages. It would be interesting to find other properties of graph nodes that can be computed similarly.

Chapter 8

A First Experience in Archiving the French Web

Abstract.

The web is a more and more valuable source of information and organizations are involved in archiving (portions of) it for various purposes, e.g., the Internet Archive www.archive.org. *A new mission of the French National Library (BnF) is the "dépôt légal" (legal deposit) of the French web. We describe here some preliminary work on the topic conducted by BnF and INRIA. In particular, we consider the acquisition of the web archive. Issues are the definition of the perimeter of the French web and the choice of pages to read once or more times (to take changes into account). When several copies of the same page are kept, this leads to versioning issues that we briefly consider. Finally, we mention some first experiments.*

Xyleme [119] supplied us with the crawling system and data sets that we used to conduct experiments. We would like to thank Mihai Preda, Gérald Sédrati, Patrick Ferran and David Le-Niniven for their contribution.

This work is in the context of important projects. The French national library and other libraries (e.g. Library of Congress) are considering building a consortium to share experience and efforts towards archiving the web. The well known foundation Internet Archive [58] also expressed interest in the present work.

8.1 Introduction

Since 1537[1], for every book edited in France, an original copy is sent to the Bibliothèque nationale de France (French National Library - BnF in short) in a process called *dépôt légal*. The BnF stores all these items and makes them available for future generations of researchers. As publication on the web increases, the BnF proposes providing a similar service for the French web, a more and more important and valuable source of information. In this chapter, we study technical issues raised by the legal deposit of the French web.

The main differences between the existing legal deposit and that of the web are the following:

1. the number of content providers: On the web, anyone can publish documents. One should compare, for instance, the 148.000 web sites in ".fr" (as of 2001) with the 5000 traditional publishers at the same date.

2. the quantity of information: Primarily because of the simplicity of publishing on the web, the size of content published on the French web is orders of magnitude larger than that of the existing legal deposit and with the popularity of the web, this will be more and more the case.

3. the quality: Lots of information on the web is not meaningful.

4. the relationship with the editors: With legal deposit, it is accepted (indeed enforced by law) that the editors "push" their publication to the legal deposit. This "push" model is not necessary on the web, where national libraries can themselves find relevant information to archive. Moreover, with the relative freedom of publication, a strictly push model is not applicable.

5. updates: Editors send their new versions to the legal deposit (again in push mode), so it is their responsibility to decide when a new version occurs. On the web, changes typically occur continuously and it is not expected that web-masters will, in general, warn the legal deposit of new releases.

6. perimeter: The perimeter of the classical legal deposit is reasonably simple, roughly *the contents published in France*. Such notion of border is more delusive on the web.

[1]This was a decision of King François the 1st.

For these reasons, the legal deposit of the French web should not only rely on editors "pushing" information to BnF. It should also involve (because of the volume of information) on complementing the work of librarians with automatic processing.

There are other aspects in the archiving of the web that will not be considered here. For instance, the archiving of sound and video leads to issues such as streaming. Also, the physical and logical storage of large amounts of data brings issues of long term preservation. How can we guarantee that several terabyte of data stored today on some storage device in some format will still be readable in 2050? Another interesting aspect is to determine which services (such as indexing and querying) should be offered to users interested in analyzing archived web content. In the present chapter, we will focus on the issue of obtaining the necessary information to properly archive the web.

We describe here preliminary works and experiments conducted by BnF and INRIA. The focus is on the construction of the web archive. This leads us to considering issues such as the definition of the perimeter of the French web and the choice of pages to read one or more times (to take changes into account). When several copies of the same page are kept, this also leads to versioning issues that we briefly consider. Finally, we mention some first experiments performed with data provided by Xyleme's crawls of the web (of close to a billion URL).

In Section 8.2, we detail the problem and mention existing work on similar topics. In Section 8.3, we consider the building of the web archive. Section 8.4 deals with the importance of pages and sites that turn out to play an important role in our approach. In Section 8.5, we discuss change representation, that is we define a notion of delta per web site that we use for efficient and consistent refresh of the warehouse. Finally we briefly present results of experiments.

8.2 Web Archiving

The web keeps growing at an incredible rate. We often have the feeling that it accumulates new information without any garbage collection and one may ask if the web is not self-archiving? Indeed, some sites provide access to selective archives. On the other hand, valuable information disappears very quickly as community and personal web pages are removed. Also the fact that there is no control of changes in "pseudo" archives is rather critical, because this leaves room for revision of history. This is why several projects aim at archiving the web. We present some of them in this section.

8.2.1 Goal and scope

The web archive intends providing future generations with a representative archive of the cultural production (in a wide sense) of a particular period of Internet history. It may be used not only to refer to well known pieces of work (for instance scientific articles) but also to provide material for cultural, political, sociological studies, and even to provide material for studying the web itself (technical or graphical evolution of sites for instance). The mission of national libraries is to archive a wide range of material because nobody knows what will be of interest for future research. This also applies to the web. But for the web, exhaustiveness, which is required for traditional publications (books, newspapers, magazines, audio CD, video, CDROM), can't be achieved. In fact, in traditional publication, publishers are actually filtering contents and an exhaustive storage is made by national libraries from this filtered material. On the web, publishing is almost free of charge, more people are able to publish and no filtering is made by the publishing apparatus. So the issue of selection comes again but it has to be considered in the light of the mission of national libraries, which is to provide future generations with a large and representative part of the cultural production of an era.

8.2.2 Similar projects

Up to now, two main approaches have been followed by national libraries regarding web archiving. The first one is to select manually a few hundred sites and choose a frequency of archiving. This approach has been taken by Australia [85] and Canada [70] for instance since 1996. A selection policy has been defined focusing on institutional and national publication.

The second approach is an automatic one. It has been chosen by Nordic countries [12] (Sweden, Finland, Norway). The use of robot crawler makes it possible to archive a much wider range of sites, a significant part of the surface web in fact (maybe 1/3 of the surface web for a country). No selection is made. Each page that is reachable from the portion of the web we know of will be harvested and archived by the robot. The crawling and indexing times are quite long and in the meantime, pages are not updated. For instance, a global snapshot of the complete national web (including national and generic domain located sites) is made twice a year by the royal library of Sweden. The two main problems with this model are: (i) the lack of updates of archived pages between two snapshots, (ii) the deep or invisible web [90, 14] that can't be harvested on line.

8.2.3 Orientation of this experiment

Considering the large amount of content available on the web, the BnF deems that using automatic content gathering method is necessary. But robots have to be adapted to provide a continuous archiving facility. That is why we have submitted a framework [71] that allows to focus either the crawl or the archiving, or both, on a specific subset of sites chosen in an automatic way. The robot is driven by parameters that are calculated on the fly, automatically and at a large scale. This allows us to allocate in an optimal manner the resources to crawling and archiving. The goal is twofold: (i) to cover a very large portion of the French web (perhaps "all", although all is an unreachable notion because of dynamic pages) and (ii) to have frequent versions of the sites, at least for a large number of sites, the most "important" ones.

It is quite difficult to capture the notion of importance of a site. An analogy taken from traditional publishing could be the number of in-going links to a site, which makes it a publicly-recognized resource by the rest of the web community. Links can be consider similar, to a certain extent of course, to bibliographical references. At least they give a web visibility to documents or sites, by increasing the probability of accessing to them (cf the random surfer in [5]). This is bringing us back to the topic of Chapter 7. We believe that it is a good analogy of the public character of traditionally published material (as opposed to unpublished, private material for instance) and a good candidate to help driving the crawling and/or archiving process [71].

The techniques of Chapter 7 have to be adapted to the context of web archiving, that is quite different. For instance, as we shall see, we have to move from a page-based notion of importance to a site-based one to build a coherent web archive (See Section 8.4). This also leads to exploring ways of storing and accessing temporal changes on sites (see Section 8.5) as we will no longer have the discrete, snapshot-type of archive but a more continuous one. To explore these difficult technical issues, a collaboration between BnF and INRIA started last year. The first results of this collaboration are presented here. Xyleme provided different sets of data needed to validate some hypothesis, using the Xyleme crawler developed jointly with INRIA. Other related issues, like the deposit and archiving of sites that can not be harvested online [72] will not be addressed here.

One difference between BnF's legal deposit and other archive projects is that it focuses on the French web. To conclude this section, we consider how this simple fact changes significantly the technology to be used.

8.2.4 The frontier for the French web

Given its mission and since others are doing it for other portions of the web, the BnF wants to focus on the French web. The notion of perimeter is relatively clear for the existing legal deposit (e.g, for books, the BnF requests a copy of each book edited by a French editor). On the web, national borders are blurred and many difficulties arise when trying to give a formal definition of the perimeter. The following criteria may be used:

- The French language. Although this may be determined from the contents of pages, it is not sufficient because of the other French speaking countries or regions e.g. Quebec. Also, many French sites now use English, e.g. there are more pages in English than in French in *inria.fr*.

- The domain name. Resource locators include a domain name that sometimes provides information about the country (e.g. *.fr*). However, this information is not sufficient and cannot in general be trusted. For instance *www.multimania.com* is hosting a large number of French associations and French personal sites and is mostly used by French people. Moreover, the registration process for *.fr* domain names is more difficult and expensive than for others, so many French sites choose other suffixes, e.g. *.com* or *.org*.

- The *address* of the site. This can be determined using information obtainable from the web (e.g., from domain name servers) such as the physical location of the web server or that of the owner of the web site name. However, some French sites may prefer to be hosted on servers in foreign countries (e.g., for economical reasons) and conversely. Furthermore, some web site owners may prefer to provide an address in exotic countries such as Bahamas to save on local taxes on site names. (With the same provider, e.g., Gandi, the cost of a domain name varies depending on the country of the owner.)

Note that for these criteria, negative information may be as useful as positive ones, e.g., we may want to exclude the domain name *.ca* (for Canada).

The Royal library of Sweden, which has been archiving the Swedish web for more than 6 years now, has settled on an inclusion policy based on national domain (.se and .nu), checking the physical address of generic domain name owners, and the possibility to manually add other sites. The distribution of the domain names they use is about 65 percent for nation domains (.se and .nu) and 25 percent for generic domains (.net, .com, .org).

Yet another difficulty in determining the perimeter is that the legal deposit is typically not very interested in commercial sites. But it is not easy to define the notion of commercial site. For instance, *amazon.fr* (note the ".fr") is commercial whereas *groups.yahoo.com/group/vertsdesevres/* (note the ".com") is a French public, political forum that may typically interest the legal deposit. As in the case of the language, the nature of web sites (e.g., commercial vs. non commercial) may be better captured using the contents of pages.

No single criteria previously mentioned is sufficient to distinguish the documents that are relevant for the legal deposit from those that are not. This leads to using a multi-criteria based clustering. The clustering is designed to incorporate some crucial information: the connectivity of the web. French sites are expected to be tightly connected. Note that here again, this is not a strict law. For instance, a French site on DNA may strongly reference foreign sites such as Mitomap (a popular database on the human mitochondrial genome).

Last but not least, the process should involve the BnF librarians and their knowledge of the web. They may know, for instance, that *00h00.com* is a web book editor that should absolutely be archived in the legal deposit.

Technical corner. The following technique is used. A crawl of the web is started. Note that sites specified as relevant by the BnF librarians are crawled first and the relevance of their pages is fixed as maximal. The pages that are discovered are analyzed for the various criteria to compute their *relevance* for the legal deposit. Only the pages believed to be relevant ("suspect" pages) are crawled. For the experiments, the OPIC algorithm (See Chapter 7) is used that allows to compute page relevance on-line while crawling the web. The algorithm focuses the crawl to portions of the web that are evaluated as relevant for the legal deposit. This is in spirit of the XML-focused on-line crawling presented in [79], except that we use the multi-criteria previously described. The technique has the other advantage that it is not necessary to store the graph structure of the web and so it can be run with very limited resources.

To conclude this section, we note that for the first experiments that we mention in the following sections, the perimeter was simply specified by the country domain name (*.fr*). We intend to refine it in the near future.

8.3 Building the Archive

In this section, we present a framework for building the archive. Previous work in this area is abundant [85, 12, 70], so we focus on the specificities of our proposal.

A simple strategy would be to take a snapshot of the French web regularly, say n times a year (based on available resources). This would typically mean running regularly a crawling process for a while (a few weeks). We believe that the resulting archive would certainly be considered inadequate by researchers. Consider a researcher interested in the French political campaigns in the beginning of the 21st century. The existing legal deposit would give him access to all issues of the *Le Monde* newspaper, a daily newspaper. On the other hand, the web archive would provide him only with a few snapshots of *Le Monde* web site per year. The researcher needs a more “real time” vision of the web. However, because of the size of the web, it would not be reasonable/feasible to archive each site once a day even if we use sophisticated versioning techniques (see Section 8.5).

So, we want some sites to be very accurately archived (almost in real-time); we want to archive a very extensive portion of the French web; and we would like to do this under limited resources. This leads to distinguishing between sites: the most important ones (to be defined) are archived frequently whereas others are archived only once in a long while (yearly or possibly never). A similar problematic is encountered when indexing the web [49]. To take full advantage of the bandwidth of the crawlers and of the storage resources, we propose a general framework for building the web archive that is based on a measure of importance for pages and of their change rate. This is achieved by adapting techniques presented in [79].

8.3.1 Site vs. page archiving

Web crawlers typically work at the granularity of pages. They select one URL to load in the collection of URLs they know of and did not load yet. The most primitive crawlers select the “first” URL, whereas the sophisticated ones select the most “important” URL [49, 79]. For an archive, it is preferable to reason at the granularity of web sites rather than just web pages. Why? If we reason at the page level, some pages in a site (more important than others) will be read more frequently. This results in very poor views of web sites. The pages of a particular site would typically be crawled at different times (possibly weeks apart), leading to dangling pointers and inconsistencies. For instance, a page that is loaded may contain a reference to a page that does not exist anymore at the time we attempt to read it or to a page whose content has been updated[2].

For these reasons, it is preferable to crawl sites and not individual pages. But it is not straightforward to define a web site. The notion of web site loosely corresponds to that of an editor for the classical legal deposit. The notion of site may be defined, as a first approximation, as the physical site name, e.g., *www.bnf.fr*. But it is not always appropriate to do so. For instance, *www.multimania.com* is the address of a web provider that hosts a large quantity of sites that we may want to archive separately. Conversely, a web site may be spread between several domain names: INRIA’s web site is on *www.inria.fr*, *www-rocq.inria.fr*, *osage.inria.fr*, *www.inrialpes.fr*, etc. There is no simple definition. For instance, people will not all agree when asked whether *www.leparisien.fr/news* and *www.leparisien.fr/shopping* are different sites or parts of the same site. To be complete, we should mention the issue of detecting mirror sites, that is very important in practice.

It should also be observed that site-based crawling contradicts compulsory crawling requirements such as the prevention of *rapid firing*. Crawlers typically balance load over many web sites to maximize bandwidth use and avoid overflooding web servers. In contrast, we focus resources on a smaller amount of web sites and try to remain at the limit of rapid firing for these sites until we have a copy of each. An advantage of this focus is that very often a small percentage of pages causes most of the problem. With site-focused crawling, it is much easier to

[2]To see an example, one of the authors (an educational experience) used, in the web site of a course he was teaching, the URL of an HTML to XML wrapping software. A few months later, this URL was leading to a pornographic web site. (Domain names that are not renewed by owners are often bought for advertisement purposes.) This is yet another motivation for archives.

detect server problems such as some dynamic page server is slow or some remote host is down.

8.3.2 Acquisition: Crawl, Discovery and Refresh

Crawl. The crawling and acquisition are based on a technique [79] that was developed at INRIA in the Xyleme project. The web data we used for our first experiments was obtained by Xyleme [119] using that technology. It allows, using a cluster of standard PCs, to retrieve a large amount of pages with limited resources, e.g. a few million pages per day per PC on average. In the spirit of [55, 59, 79], pages are read based on their importance and refreshed based on their importance and change frequency rate. This results in an optimization problem that is solved with a dynamic algorithm that was presented in [79]. The algorithm has to be adapted to the context of the web legal deposit and site-based crawling.

Discovery. We first need to allocate resources between the discovery of new pages and the refreshing of already known ones. For that, we proceed as follows. The size of the French web is estimated roughly. In a first experiment using only ".fr" as criteria and a crawl of close to one billion of URLs, this was estimated to be about 1-2 % of the global web, so of the order of 20 millions URLs. Then the librarians decide the portion of the French web they intend to store, possibly all of it (with all precautions for the term "all"). It is necessary to be able to manage in parallel the discovery of new pages and the refresh of already read pages. After a stabilization period, the system is aware of the number of pages to read for the first time (known URLs that were never loaded) and of those to refresh.

It is clearly of interest to the librarians to have a precise measure of the size of the French web. At a given time, we have read a number of pages and some of them are considered to be part of the French web. We know of a much greater number of URLs, of which some of them are considered "suspects" for being part of the French web (because of the ".fr" suffix or because they are closely connected to pages known to be in the French web, or for other reasons.) This allows us to obtain a reasonably precise estimate of the size of the French web.

Refresh. Now, let us consider the selection of the next pages to refresh. The technique used in [79] is based on a cost function for each page, the penalty for the page to be stale. For each page p, $cost(p)$ is proportional to the importance of page $i(p)$ and depends on its estimated change frequency $ch(p)$. We define in the next subsection the importance $i(S)$ of a site S and we also need to define the "change rate" of a site. When a page p in site S has changed, the site has changed. The change rate is, for instance, the number of times a page changes per year. Thus, the upper bound for the change rate of S is $ch(S) = \sum_{p\ in\ S}(ch(p))$. For efficiency reasons, it is better to consider the average change rate of pages, in particular depending on the importance of pages. We propose to use a weighted average change rate of a site as:

$$\bar{ch}(S) = \frac{\sum_p ch(p) * i(p)}{\sum_p i(p)}$$

Our refreshing of web site is based on a cost function. More precisely, we choose to read next the site S with the maximum ratio:

$$\rho(S) = \frac{\theta(i(S), \bar{ch}(S), lastCrawl(S), currentTime)}{\text{number of pages in } S}$$

where θ may be, for instance, the following simple cost function:

$$\theta = i(S) * (currentTime - lastCrawl(S)) * \bar{ch}(S)$$

We divide by the number of pages to take into account the cost to read the site. A difficulty for the first loading of a site is that we do not know for new sites their number of pages. This has to be estimated based on the number of URLs we know of the site (and never read). Note that this technique forces us to compute importance at page level.

In next Section, we revisit the notion of importance, and we propose using a notion of importance at web site level.

8.4 New notions of importance for web archiving

In this section, we extend the notion of page importance in two directions. The first one is to consider not only the graph of the web, but the content of each page. The second one is to consider the notion of importance at web site level rather than at page level.

8.4.1 Importance of pages for the legal deposit

When discovering and refreshing web pages, we want to focus on those which are of interest for the legal deposit. The classical notion of importance is used. But it is biased to take into account the perimeter of the French web. Finally, the content of pages is also considered. A librarian typically would look at some documents and know whether they are interesting. We would like to perform such an evaluation automatically, to some extent. More precisely, we can use for instance the following simple criteria:

- **Frequent use of infrequent Words:** The frequency of words found in the web page is compared to the average frequency of such words in the French web[3]. For instance, for a word w and a page p, it is:

  $$I_w = \sum_{each\ word} \frac{f_{p,w}}{f_{web}} \quad \text{where} \;\; f_{p,w} = n_{p,w}/N_p$$

  and $n_{p,w}$ is the number of occurrences of a word w in a page and N_p the number of words in the page. Intuitively, it aims at finding pages dedicated to a specific topic, e.g. butterflies, so pages that have some content.

- **Text Weight:** This measure represents the proportion of text content over other content like HTML tags, product or family names, numbers or experimental data. For instance, one may use the number of bytes of French text divided by the total number of bytes of the document.

  $$I_{pt} = \frac{size_{french\ words}}{size_{doc}}$$

  Intuitively, it increases the importance of pages with text written by people versus data, image or other content.

A first difficulty is to evaluate the relevance of these criteria. Experiments are being performed with librarians to understand which criteria best match their expertise in evaluating sites. Another difficulty is to combine the criteria. For instance, *www.microsoft.fr* may have a high PageRank, may use frequently some infrequent words and may contain a fair proportion of text. Still, due to its commercial status, it is of little interest for the legal deposit. Note that librarians are

[3]To guarantee that the most infrequent words are not just spelling mistake, the set of words is reduced to words from a French dictionary. Also, as standard, stemming is used to identify words such as *toy* and *toys*.

vital in order to "correct" errors by positive action (e.g., forcing a frequent crawl of *00h00.com*) or negative one (e.g., blocking the crawl of *www.microsoft.fr*). Furthermore, librarians are also vital to correct the somewhat brutal nature of the construction of the archive. Note however that because of the size of the web, we should avoid as much as possible manual work and would like archiving to be as fully automatic as possible.

As was shown in this section, the quality of the web archive will depend on complex issues such as being able to distinguish the borders of a web site, analyze and evaluate its content. There are ongoing projects like THESU [54] which aim at analyzing thematic subsets of the web using classification, clustering techniques and the semantics of links between web pages. Further work on the topic is necessary to improve site discovery and classification

To conclude this section, we need to extend previously defined notions to the context of web site. For some, it suffices to consider the site as a huge web document and aggregate the values of the pages. For instance, for *Frequent use of infrequent Words,* one can use:

$$I_w = \sum_{each\ word} \frac{f_{site}}{f_{web}} \quad \text{where} \quad f_{S,w} = \sum_{p\ in\ S}(n_{p,w}) / \sum_{p\ in\ S}(N_p)$$

Indeed, the values on word frequency and text weight seem to be more meaningful at the site level than at the page level.

For page importance, it is difficult. This is the topic of next section.

8.4.2 Site-based Importance

Observe that the notion of page importance is becoming less reliable as the number of dynamic pages increases on the web. A reason is that the semantics of the web graph created by dynamic pages is weaker than the previous document based approach. Indeed, dynamic pages are often the result of database queries and link to other queries on the same database. The number of incoming/outgoing links is now related to the size of the database and the number of queries, whereas it was previously a human artifact carrying stronger semantics. In this section, we present a novel definition of sites' importance that is closely related to the already known page importance. The goal is to define a site importance with stronger semantics, in that it does not depend on the site internal databases and links. We will see how we can derive such importance from this site model.

To obtain a notion of site importance from the notion of page importance, one could consider a number of alternatives:

- Consider only links between web sites and ignore internal links;
- Define site importance as the sum of PageRank values for each page of the web site;
- Define site importance as the maximum value of PageRank, often corresponding to that of the site main page.

Page importance, namely PageRank in Google terminology, is defined as the fixpoint of the matrix equation $X = L * X$ [16, 88], where the web-pages graph G is represented as a link matrix $L[1..n, 1..n]$. Let $out[1..n]$ be the vector of out-degrees. If there is an edge for i to j, $L[i,j] = 1/out[i]$, otherwise it is 0. We note $I_{page}[1..n]$ the importance for each page. Let us define a web-sites graph G' where each node is a web-site (e.g. *www.inria.fr*). The number of web-sites is n'. For each link from page p in web-site Y to page q in web-site Z there is an edge from Y to Z. This edges are weighted, that is if page p in site S is twice more important than page p' (in S also), then the total weight of outgoing edges from p will be twice the total weight of outgoing edges from p'. The obvious reason is that browsing the web remains page based, thus links coming from more important pages deserve to have more weight than links coming from less important ones. The intuition underlying these measures is that a web observer will visit randomly each page proportionally to its importance. Thus, the link matrix is now defined by:

$$L'[Y,Z] = \sum_{p\ in\ Y,\ q\ in\ Z} \frac{I_{page}[p]}{\sum_{p'\ in\ Y} I_{page}[p']} * L[p,q]$$

We note two things:

- If the graph G representing the web-graph is (artificially or not) strongly connected, then the graph G' derived from G is also strongly connected.
- L' is still a stochastic matrix, in that $\forall Y, \sum_Z L'[Y,Z] = 1$. (proof in appendix).

Thus, the page importance, namely PageRank, can be computed over G', L' and there is a unique fixpoint solution. We prove in Appendix B that the solution is given by:

$$I_{site}[Y] = \sum_{p\ in\ Y} I_{page}[p]$$

This formal relation between web site based importance and page importance suggests to compute page importance for all pages, a rather costly task. However, it serves as a reference to define site-based importance, and helps understand its relation to page-based importance. One could simplify the problem by considering, for instance, that all pages in a web site have the same importance. Based on this, the computation of site-importance becomes much simpler. In this case, if there is there is at least one page in Y pointing to one page in Z, we have $L'[Y, Z] = 1/out(Y)$, where $out(Y)$ is the out-degree of Y. A more precise approximation of the reference value consists in evaluating the importance of pages of a given web site S on the restriction of G to S. Intuitively it means that only internal links in S will be considered. This approximation is very effective because: (i) it finds very good importance values for pages, that correspond precisely to the internal structure of the web-site (ii) it is cheaper to compute the internal page importance for all web sites, one by one, than to compute the PageRank over the entire web (iii) the semantics of the result are stronger because it is based on site-to-site links.

This web-site approach enhances significantly previous work in the area, and we will see in next section how we also extend previous work in change detection, representation and querying to web sites.

8.5 Representing Changes

Intuitively, change control and version management are used to save storage and bandwidth resources by updating in a large data warehouse only the small parts that have changed [69]. We want to maximize the use of bandwidth, for instance, by avoiding the loading of sites that did not change (much) since the last time they were read. To maximize the use of storage, we typically use compression techniques and a clever representation of changes. We propose in this section a change representation at the level of web sites in the spirit of [63, 69] (See Chapter 4). Our change representation consists of a *site-delta*, in XML, with the following features:

(i) Persistent identification of web pages using their URL, and unique identification of each document using the tuple (URL, date-of-crawl);

(ii) Information about mirror sites and their up-to-date status;

(iii) Support for temporal queries and browsing the archive

The following example is a portion of the *site-delta* for *www.inria.fr*:

```
<website url="www.inria.fr">
<page url="/index.html">
  <document date="2002-Jan-01" status="updated"
          file="543B6.html"/>
  <document date="2002-Mar-01" status="unchanged"
          file="543B6.html"/>
</page>
<page url="/news.html">
  <document date="2002-Mar-25" status="updated"
          file="543GX6.html"/>
  <document date="2002-Mar-24" status="error">
    <error httperror="404"/>
  </document>
  <document date="2002-Mar-23" status="updated"
          file="523GY6.html"/>
  ...
  <document date="1999-Jan-08" status="new"
          file="123GB8.html"/>
</page>
<mirror url="www-mirror.inria.fr" depth="nolimit">
  <exclusion path="/cgi-bin" />
</mirror>
</website>
```

Each web-site element contains a set of pages, and each page element contains a subtree for each time the page was accessed. If the page was successfully retrieved, a reference to the archive of the document is stored, as well as some metadata. If an error was encountered, the page status is updated accordingly. If the page mirrors another page on the same (or on another) web-site, the document is stored only once (if possible) and is tagged as a mirror document. Each web-site tree also contains a list of web-sites mirroring part of its content. The up-to-date status of mirror sites is stored in their respective XML file.

Other usages. The site-delta is not only used for storage. It also improves the efficiency of the legal deposit. In particular, we mentioned previously that the legal deposit works at a site level. Because our site-delta representation is designed to maintain information at page level, it serves as an intermediate layer between site-level components and page-based modules.

For instance, we explained that the acquisition module crawls sites instead of pages. The site-delta is then used to provide information about pages (last update, change frequency, file size) that will be used to reduce the number of pages to crawl by using caching strategies. Consider a news web site, e.g. *www.leparisien.fr/*. News articles are added each day and seldom modified afterwards, only the index page is updated frequently. Thus, it is not desirable to crawl the entire web site every day. The site-delta keeps track of the metadata for each pages and allows to decide which pages should be crawled. So it allows the legal deposit to virtually crawl the entire web site each day.

Browsing the archive. A standard first step consists in replacing links to the Internet (e.g. *http://www.yahoo.fr/*) by local links (e.g. to files). The process is in general easy, some difficulties are caused by pages using java-scripts (sometimes on purpose) that make links unreadable. A usual problem is the consistency of the links and the data. First, the web graph is not consistent to start; broken links, servers down, pages with out of date data are common. Furthermore, since pages are crawled very irregularly, we never have a true snapshot of the web.

The specific problem of the legal deposit is related to *temporal browsing*. Consider, for instance, a news web site that is entirely crawled every day. A user may arrive at a page, perhaps via a search engine on the archive. One would expect to provide him the means to browse through the web site of that day and also in time, move to this same page the next day. The problem becomes seriously more complex when we consider that all pages are not read at the same time. For instance, suppose a user reads a version t of page p and clicks on a link to p'. We may not have the value of page p' at that time. Should we find the latest version of p' before t, the first version after t, or the closest one? Based on an evaluation of the change frequency of p', one may compute which is the most likely to be the correct one. However, the user may be unsatisfied by this and it may be more appropriate to propose several versions of that page.

One may also want to integrate information coming from different versions of a page into a single one. For instance, consider the index of a news web site with headlines for each news article over the last few days. We would like to *automatically* group all headlines of the week into a single index page, as in Google news search engine [50]. A difficulty is to understand the structure of the document and to select the valuable links. For instance, we probably don't want to group all advertisements of the week!

8.6 Conclusion

As mentioned in the introduction, this chapter describes preliminary work. Some experiments have already been conducted. A crawl of the web was performed and data has been analyzed by BnF librarians. In particular, the goal was to evaluate the relevance of page importance (i.e., PageRank in Google terminology). This notion has been validated, to a certain extent, by the success of search engines that use it. It was not clear whether it is adapted to web archiving. First results seem to indicate that the correlation between automatic ranking and that of librarians is essentially as similar as the correlation between ranking by librarians. These results have been briefly presented in Chapter 7.

Perhaps the most interesting aspect of this archiving work is that it leads us to reconsider notions such as web site or web importance. We believe that this is leading to a better understanding of the web. We intend to pursue this line of study and try to see how to take advantage of techniques in classification or clustering. Conversely, we intend to use some of the technology developed here to guide the classification and clustering of web pages.

Part III

Conclusion

Conclusion

In this thesis, we presented our work on the topic of change-control (i) for semi-structured data, (ii) in the context of the web. There are two significantly different approaches, we have seen that the first corresponds to a microscopic-scale analysis of documents from the web, whereas the second corresponds to a macroscopic-scale analysis of the web. However, we discovered through their study that the two approaches are strongly tied together:

- Quantity of data. The Internet contains huge amounts of data, but the data often consists in collections of small documents. Thus, the quantitative approach consists in a local analysis of data, but it relies of the fact that each piece of data is part of a larger set of data. This is necessary to improve the performance of handling multiple pieces of data, and to scale up to the web data size. Thus, efficiency, performance, memory and storage management remain a key aspect of all systems.

- Quality of data. The semantic analysis of data also leads to connecting the microscopic and macroscopic scales. Indeed, the semantic analysis consists in finding a semantic value and interpretation for each small piece of data, but in the context of the global data warehouse (or knowledge center) that is the web. This notion of context can be seen for instance through the use of DTDs (or XMLSchema): each XML fragments is analyzed separately but in the context of a global schema. The importance of the global schema may also be seen as relying on a human or social semantic: for instance the importance of web pages depends on the way authors consider each web page they know of. Thus, the microscopic analysis of documents (e.g. page links) enables the constructions of a macroscopic knowledge such as page importance. Conversely, the global knowledge improves the semantic analysis and semantical interpretation of pieces of local data.

The main part of our work consisted in proposing algorithms, their implementations and experiments to validate them. More precisely, the algorithms we proposed are as follows:

- an algorithm to compute the differences between XML files,
- a formal model and some algorithms to manage change operations between XML documents,
- an algorithm to compute online the page importance on the web, a proof of the convergence of this algorithm, and an extension to support dynamic graphs (i.e. the changing web) of this algorithm,
- a framework and algorithms that can be used in the context of the web archiving, and in particular of choosing web sites to archive.

Moreover, we contributed to broadening existing knowledge with, for instance, a comparative study of change detection algorithms in XML, and XML change representation formats.

This work has been published in international conferences as well as in French community conferences. A list is presented in Appendix A. [69, 84, 35, 4, 5]. All algorithms presented here have been implemented, in the context of important projects such as [34, 117]. Most of our work has been transfered to industry, in particular to Xyleme [119], or is available as open-source freeware [34].

Through our work, we noted the lack of a precise model and framework for change-control on the web, both at the scale of documents and at the scale of the Internet. Indeed, there are at this time no mechanisms that enable an application to manage changes that occur on the web. While it is possible for a user to subscribe to the notification system of some web sites, there is no global framework that permits the management of changes all over the web. There is not enough information for automatic surveillance, neither in documents (e.g. reliable HTTP headers informations, or HTML/XML metadata), nor at the scale of the web (e.g. servers that have informations on the versions of collections of documents).

In a similar way, we noted the lack of a standard for change-control in XML documents. Some interesting formats have been proposed, but more work is necessary to obtain a precise framework and change model for managing temporal and historical data.

For part of it, these issues (e.g. storage and querying of changes) will probably be solved in the future. This will lead to the development of better services for the users of the web.

However, I am afraid that the lack of precise tools for change management may be due to the notion of *changes* itself. Isn't there an almost philosophical issue to ask whether changes can be managed, while the meaning of changes is to make things different that what we already have? In the context of the SPIN project for instance, we somehow faced that problem. The SPIN project consists in managing a set of web documents of interest for the users. The documents are changing, so is our warehouse, and web services are also enriching the knowledge contained in our warehouse. We tried to develop change-control mechanism, but we realized how difficult it was to develop a generic application for interpreting changes. This difficult problem somehow relates to "belief revision", the introduction of changes in logical theories [82, 95].

As a consequence, change-applications should manipulate change items in a framework that is precise and does not change itself. It is then possible to process and understand the changes of the items.

It is a challenge to develop and promote a framework that enables the efficient management of changing data on the web, in the spirit (and with the success) of XML and Web Services that are used today to exchange and store data.

Part IV

Appendix

Appendix A

List of Publications

Journals

- Computing web page importance without storing the graph of the web (extended abstract), *Serge Abiteboul, Mihai Preda, Gregory Cobena*, IEEE Data Engineering Bulletin, Volume 25, March 2002

- A dynamic warehouse for XML data of the Web, *Lucie Xyleme (I am one of the 25 authors)*, IEEE Data Engineering Bulletin, June 2001.

Program Committees

- Organizer and PC-Member of 3rd ECDL Workshop on Web Archives, Trondheim, Norway, 2003 (to come).

International Conferences

- Adaptive Online Page Importance Computation, *Serge Abiteboul and Mihai Preda and Gregory Cobena*, WWW 2003 (Budapest, Hungary)

- Dynamic XML Documents with Distribution and Replication, *Serge Abiteboul and Angela Bonifati and Gregory Cobena and Ioana Manolescu and Tova Milo*, SIGMOD 2003 (San Diego, USA)

- A First Experience in Archiving the French Web, *Serge Abiteboul, Gregory Cobena, Julien Masanes, Gerald Sedrati*, ECDL 2002 (Rome, Italy)

- Detecting Changes in XML Documents, *Gregory Cobena, Serge Abiteboul, Amélie Marian*, ICDE 2002 (San Jose, USA)
- Change-Centric Management of Versions in an XML Warehouse, *Amélie Marian, Serge Abiteboul, Gregory Cobena, Laurent Mignet*, VLDB 2001 (Rome, Italy).
- Monitoring XML data on the Web, *Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, Mihai Preda*, SIGMOD 2001 (Santa Barbara, USA).

Workshops and French Conferences

- A comparative study for XML change detection, *Grégory Cobena, Talel Abdessalem, Yassine Hinnach*, BDA 2002 (Evry, France). Currently submitted to TKDE Journal.
- Construction and Maintenance of a Set of Pages of Interest (SPIN), *Serge Abiteboul, Grégory Cobena, Benjamin Nguyen, Antonella Poggi*, BDA 2002 (Evry, France)
- Detecting Changes in XML Documents, *Gregory Cobena, Serge Abiteboul, Amélie Marian*, BDA 2001 (Agadir, Maroc) (also in ICDE 2002)
- Querying Subscription in an XML Web-house, *Benjamin Nguyen, Serge Abiteboul, Grégory Cobena, Laurent Mignet*, on First DELOS Workshop on Digital Libraries 2000.

Unpublished

- Crawling important sites on the Web, *Gregory Cobena, Serge Abiteboul*, 2nd ECDL Workshop on Web Archiving (Rome, Italy)
- A comparative study for XML change detection, *Grégory Cobena, Talel Abdessalem, Yassine Hinnach*, Currently submitted to TKDE Journal.

Appendix B

Site-Based Importance - Proof

In this appendix, we give the details of our model of site-based importance.

Lemma B.0.1
Let $L[1..n, 1..n]$ be the link matrix of web pages and $I[1..n]$ the vector of importance for pages. If there is an edge for i to j, $L[i,j] = 1/out[i]$, otherwise $L[i,j] = 0$. The graph G corresponding to L is strongly connected and a-periodic. L is stochastic. Then L' defined as

$$L'[Y,Z] = \sum_{p\ in\ Y,\ q\ in\ Z} \frac{I_{page}[p]}{\sum_{p'\ in\ Y} I_{page}[p']} * L[p,q]$$

is also stochastic. We also note that if G is strongly connected, then the graph G' corresponding to L' is also strongly connected.

Proof:

$$\sum_{Z} L'[Y,Z] = \sum_{Z} \sum_{p\ in\ Y} \frac{I_{page}[p]}{\sum_{p'\ in\ Y} I_{page}[p']} * \sum_{q\ in\ Z} L[p,q]$$

$$\sum_{Z} L'[Y,Z] = \sum_{p\ in\ Y} \frac{I_{page}[p]}{\sum_{p'\ in\ Y} I_{page}[p']} * (\sum_{Z} \sum_{q\ in\ Z} L[p,q])$$

$$\sum_{Z} L'[Y,Z] = \sum_{p\ in\ Y} \frac{I_{page}[p]}{\sum_{p'\ in\ Y} I_{page}[p']} * 1$$

$$\sum_{Z} L'[Y,Z] = 1$$

Theorem B.0.1
Let $I_{site}[Y] = \sum_{p\ in\ Y} I_{page}[p]$. Then I_{site} is the (unique) value of page importance corresponding to the graph G' and link matrix L'.

Proof: The proof is derived from the formalism presented in our on-line page importance work [5] (see Chapter 7). In this formalism, when a page is read, we distribute its "cash" its children. The average speed at which a page receives "cash" from other pages corresponds exactly to its importance. It is measure by $H[i]$, the accumulation of cash. In a summary, we have that $H[i] = \sum_j H[j] * L[i,j]$, so that the normalization of H converges to page importance I.

Now let us apply the exact same algorithm, i.e. at page level, but let us focus on the "cash" moving from a web site to another. Let the history at site level be the sum of the history of its pages. So:

$$H_{site[Y]} = \sum_{p\ in\ Y} H[p]$$

.

Now consider the "cash" distributed by site Y to a site Z. It is the sum of the "cash" distributed by the pages of Y to pages of Z. That is:

$$\sum_{p\ in\ Y} \sum_{q\ in\ Z} H[p] * L[p,q]$$

. Which can also be written as $\sum_{p\ in\ Y} H[p] \sum_{q\ in\ Z} \frac{H[p]}{H[Y]} * L[p,q]$, or $H[Y] * L'[Y,Z]$. Thus,

$$H[Z] = \sum_{Y} H[Y] * L'[Y,Z]$$

, by the definition of L'.

As a consequence, the importance I_{site} defined previously is a solution of the on-line formalism applied to graph G'. Our formalism [5] is proved to be strictly equivalent to PageRank. Moreover, the matrix convergence solution is unique because G' is strongly connected. Thus, I_{site} is exactly the PageRank over G'.

Bibliography

[1] S. Abiteboul, O. Benjelloun, Ioana Manolescu, Tova Milo, and Roger Weber. Active XML: Peer-to-Peer Data and Web Services Integration (demo). In *VLDB*, 2002.

[2] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. In *SIGMOD*, 2003.

[3] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publisher, October 1999.

[4] S. Abiteboul, G. Cobena, J. Masanes, and G. Sedrati. A first experience in archiving the french web. *ECDL*, 2002.

[5] S. Abiteboul, M. Preda, and G. Cobena. Computing web page importance without storing the graph of the web (extended abstract). *IEEE-CS Data Engineering Bulletin, Volume 25*, 2002.

[6] Serge Abiteboul. Le dépôt légal du web, terrain de compétition à la française. In *Le Monde*, Vendredi 5 avril 2002.
`http://www-rocq.inria.fr/ abite-`
`bou/pub/lemonde02.html` .

[7] Serge Abiteboul, Mihai Preda, and Gregory Cobena. Adaptive On-Line Page Importance Computation. In *WWW12*, 2003.

[8] V. Aguiléra, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying XML Documents in Xyleme. In *Proceedings of the ACM-SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, july 2000.

[9] Alta vista.
`http://www.altavista.com/` .

[10] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *The VLDB Journal*, pages 53–64, 2000.

[11] A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*. Oxford University Press, 1997.

[12] Allan Arvidson, Krister Persson, and Johan Mannerheim. The kulturarw3 project - the royal swedish web archiw3e - an example of 'complete' collection of web pages. In *66th IFLA Council and General Conference*, 2000. `www.ifla.org/IV/ifla66/papers/154-157e.htm`.

[13] David T. Barnard, Gwen Clarke, and Nicolas Duncan. Tree-to-tree Correction for Document Trees David T. Barnard.

[14] M.K. Bergman. The deep web: Surfacing hidden value. `www.brightplanet.com/` .

[15] K. Bharat and A. Broder. Estimating the relative size and overlap of public web search engines. *7th International World Wide Web Conference (WWW7)*, 1998.

[16] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *WWW7 Conference, Computer Networks 30(1-7)*, 1998.

[17] Andrei Z. Broder and al. Graph structure in the web. *WWW9/Computer Networks*, 2000.

[18] Peter Buneman, Susan Davidson, Wenfei Fan, Carmen Hara, and Wang-Chiew Tan. Keys for XML. *Computer Networks, Vol. 39*, August 2002.

[19] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan. Archiving Scientific Data. In *SIGMOD*, 2002.

[20] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: a new approach to topic-specific web resource discovery. *8th World Wide Web Conference*, 1999.

[21] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *ICDE*, 2002.

[22] S. Chawathe. Comparing Hierarchical Data in External Memory. In *VLDB*, 1999.

[23] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *ICDE*, 1998.

[24] S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *SIGMOD*, pages 26–37, Tuscon, Arizona, May 1997.

[25] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD*, 25(2):493–504, 1996.

[26] S. S. Chawathe, S. Abiteboul, and J. Widom. Managing Historical Semi-structured Data. *Theory and Practice of Object Systems*, 5(3):143–162, August 1999.

[27] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *SIGMOD*, 2000.

[28] S. Chien, V. Tsotras, and C. Zaniolo. A Comparative Study of Version Management Schemes for XML Documents. TimeCenter Technical Report TR51, Sept. 2000.

[29] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Version management of XML documents. *Lecture Notes in Computer Science*, 2001.

[30] Steve Chien, Cynthia Dwork, Ravi Kumar, Dan Simon, and D. Sivakumar. Link evolution: Analysis and algorithms. In *Workshop on Algorithms and Models for the Web Graph (WAW)*, 2002.

[31] Junghoo Cho, Hector García-Molina, and Lawrence Page. Efficient crawling through URL ordering. *Computer Networks and ISDN Systems*, 30(1–7):161–172, 1998.

[32] Kai Lai Chung. Markov chains with stationary transition probabilities. *Springer*, 1967.

[33] J. Clark. XT, a Java implementation of XSLT.
`http://www.jclark.com/xml/xt.html` .

[34] G. Cobéna, S. Abiteboul, and A. Marian. XyDiff, tools for detecting changes in XML documents.
`http://www-rocq.inria.fr/~cobena/XyDiffWeb/` .

[35] Grégory Cobéna, Serge Abiteboul, and Amélie Marian. Detecting changes in xml documents. In *ICDE*, 2002.

[36] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *PODS*, 2002.

[37] Colin Cooper and Alan M. Frieze. A general model of undirected web graphs. In *European Symposium on Algorithms*, pages 500–511, 2001.

[38] F.P. Curbera and D.A. Epstein. Fast difference and update of xml documents. In *XTech*, 1999.

[39] CVS, Concurrent Versions System.
`http://www.cvshome.org` .

[40] Sankoff D. and J. B. Kruskal. Time warps, string edits, and macromolecules. *Addison-Wesley, Reading, Mass.*, 1983.

[41] Y. Dong D. Zhang. An efficient algorithm to rank web resources. *9th International World Wide Web Conference*, 2000.

[42] J. Dean and M.R. Henzinger. Finding related pages in the world wide web. *8th International World Wide Web Conference*, 1999.

[43] Decision Soft. XML diff.
`http://tools.decisionsoft.com/xmldiff.html`.

[44] DeltaXML. Change control for XML in XML.
`http://www.deltaxml.com/`.

[45] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML.
`http://www.w3.org/TR/NOTE-xml-ql/` .

[46] Dommitt Inc. XML diff and merge tool.
`www.dommitt.com`.

[47] FSF. GNU diff.
`www.gnu.org/software/diffutils/diffutils.html` .

[48] F. R. Gantmacher. Applications of the theory of matrices. In *Interscience Publishers*, pages 64–79, 1959.

[49] Google.
`http://www.google.com/` .

[50] Google. Google news search.
`http://news.google.com/` .

[51] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB*, 2002.

[52] G. Gottlob, C. Koch, and R. Pichler. XPath Query Evaluation: Improving Time and Space Efficiency. In *ICDE*, 2003.

[53] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata. In *ICDT*, 2003.

[54] Maria Halkidi, Benjamin Nguyen, Iraklis Varlamis, and Mihalis Vazirgianis. Thesus: Organising web document collections based on semantics and clustering. Technical Report, 2002.

[55] T. Haveliwala. Efficient computation of pagerank. *Technical report, Stanford University*, 1999.

[56] IBM. XML Treediff.
`http://www.alphaworks.ibm.com/` .

[57] Oasis, ice resources.
`www.oasis-open.org/cover/ice.html` .

[58] Internet archive.
`http://www.archive.org/` .

[59] H. Garcia-Molina J. Cho. Synchronizing a database to improve freshness. *SIGMOD*, 2000.

[60] T. Jiang, L. Wang, and Kaizhong Zhang. Alignement of Trees - An Alternative to Tree Edit. *Proceedings of the Fifth Symposium on Combinatorial Pattern Matching, pp.75-86*, 1994.

[61] Kinecta.
`http://www.kinecta.com/products.html` .

[62] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[63] Robin La Fontaine. A Delta Format for XML: Identifying changes in XML and representing the changes in XML. In *XML Europe*, 2001.

[64] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *VLDB*, Bombay, India, 1996.

[65] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory 10*, pages 707–710, 1966.

[66] Michael Ley. Dblp.
`http://dblp.uni-trier.de/` .

[67] Logilab. XML diff.
`http://www.logilab.org/xmldiff/` .

[68] S. Lu. A tree-to-tree distance and its application to cluster analysis. *in IEEE Transaction on Pattern Analysis and Machine Intelligence, Vol. 2*, 1979.

[69] Amélie Marian, Serge Abiteboul, Grégory Cobéna, and Laurent Mignet. Change-centric Management of Versions in an XML Warehouse. *VLDB*, 2001.

[70] L. Martin. Networked electronic publications policy, 1999.
`www.nlc-bnc.ca/9/2/p2-9905-07-f.html` .

[71] J. Masanès. The BnF's project for web archiving. In *What's next for Digital Deposit Libraries? ECDL Workshop*, 2001.
`www.bnf.fr/pages/infopro/ecdl/france/sld001.htm` .

[72] J. Masanès. Préserver les contenus du web. In *IVe journées internationales d'études de l'ARSAG - La conservation à l'ère du numérique*, 2002.

[73] Julien Masanès. Towards Continuous Web Archiving: First Results and an Agenda for the Future. In *D-Lib*, 2002.
`http://www.dlib.org/` .

[74] W.J. Masek and M.S. Paterson. A faster algorithm for computing string edit distances. In *J. Comput. System Sci.*, 1980.

[75] G.V. Meghabghab. Google's web page ranking applied to different topological web graph structures. *JASIS 52(9)*, 2001.

[76] Microsoft. XDL: XML Diff Language.
`http://www.gotdotnet.com/team/xmltools/xmldiff/` .

[77] Microsoft. XML Diff and Patch.
`http://www.gotdotnet.com/team/xmltools/xmldiff/` .

[78] Sun Microsystems. Making all the difference.
`http://www.sun.com/xml/developers/diffmk/` .

[79] L. Mignet, M. Preda, S. Abiteboul, S. Ailleret, B. Amann, and A. Marian. Acquiring XML pages for a WebHouse. In *proceedings of Base de Données Avancées conference*, 2000.

[80] Rajeev Motwani and Prabhakar Raghavan. Randomized algorithms. *ACM Computing Surveys*, 28(1):33–37, 1996.

[81] Eugene W. Myers. An O(ND) difference algorithm and its variations. In *Algorithmica*, 1986.

[82] Bernhard Nebel. A knowledge level analysis of belief revision. In R. Brachman, H. J. Levesque, and R. Reiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 1st International Conference*, pages 301–311, San Mateo, 1989. Morgan Kaufmann.

[83] B. Nguyen and S. Abiteboul. A hash-tree based algorithm for subset detection: analysis and experiments. In *Tech. Report*, 2002.
`www-rocq.inria.fr/verso/`.

[84] B. Nguyen, S. Abiteboul, G. Cobéna, and M. Preda. Monitoring XML Data on the Web. In *SIGMOD*, 2001.

[85] A National Library of Australia Position Paper. National strategy for provision of access to australian electronic publications. `www.nla.gov.au/policy/paep.html` .

[86] Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. XPath: Looking Forward. In *LNCS, Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490, pages 109–127. Springer, 2002.

[87] OMG: Object Management Group - CORBA and IDL. `http://www.omg.org/` .

[88] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web, 1998.

[89] M. Preda. Data acquisition for an xml warehouse. *DEA thesis Paris 7 University*, 2000.

[90] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *The VLDB Journal*, 2001.

[91] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). `http://www.w3.org/TandS/QL/QL98`.

[92] L. Giles S. Lawrence. Accessibility and distribution of information on the web. *Nature*, 1999.

[93] SAX Simple API for XML. `http://www.saxproject.org/` .

[94] Search-engine watch. `www.searchenginewatch.com/` .

[95] Richard Segal. Belief revision, 1994.

[96] Luc Segoufin and Victor Vianu. Validating Streaming XML Documents. In *PODS*, 2002.

[97] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters, 6*, pages 184–186, 1977.

[98] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms, 11*, pages 581–621, 1990.

[99] K.C. Tai. The tree-to-tree correction problem. In *Journal of the ACM, 26(3)*, pages 422–433, july 1979.

[100] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6), 1997.

[101] C.J. van Rijsbergen. Information retrieval. *London, Butterworths*, 1979.

[102] W3C. EXtensible Markup Language (xml) 1.0.
`http://www.w3.org/TR/REC-xml` .

[103] W3C. Rdf, resource description framework.
`http://www.w3.org/RDF` .

[104] W3c: World wide web consortium.
`http://www.w3.org/` .

[105] W3Schools Online Web Tutorials.
`http://www.w3schools.com/` .

[106] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Jour. ACM 21*, pages 168–173, 1974.

[107] Jason T.L. Wang, Bruce A. Shapiro, Dennis Shasha, Kaizhong Zhang, and Kathleen M. Currey. An algorithm for finding the largest approximately common substructures of two trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):889–895, 1998.

[108] Jason T.L. Wang, Kaizhong Zhang, and Dennis Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):559–571, 1994.

[109] Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X-diff: A fast change detection algorithm for xmldocuments. In *ICDE*, 2003.
`http://www.cs.wisc.edu/ yuanwang/xdiff.html` .

[110] N. Webber, C. O'Connell, B. Hunt, R. Levine, L.Popkin, and G. Larose. The Information and Content Exchange (ICE) Protocol.
`http://www.w3.org/TR/NOTE-ice` .

[111] S. Wu, U. Manber, and G. Myers. An O(NP) sequence comparison algorithm. In *Information Processing Letters*, pages 317–323, 1990.

[112] Xalan Java version 2.2D11.
`http://xml.apache.org/xalan-j/` .

[113] Xerces C++, Initial code base derived from IBM's XML4C Version 2.0.
`http://xml.apache.org/xerces-c/` .

[114] XML DB. Xupdate.
`http://www.xmldb.org/xupdate/` .

[115] XML-TK: An XML Toolkit for Light-weight XML Stream Processing.
`http://www.cs.washington.edu/homes/suciu/XMLTK/` .

[116] XPath, xml path language.
`http://www.w3.org/TR/xpath/` .

[117] Xyleme Project.
`http://www-rocq.inria.fr/verso/`.

[118] Lucie Xyleme. A dynamic warehouse for xml data of the web. *IEEE Data Engineering Bulletin*, 2001.

[119] Xyleme.
`www.xyleme.com`.

[120] Tak W. Yan and Hector Garcia-Molina. The SIFT Information Dissemination System. In *TODS 24(4): 529-565*, 1999.

[121] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience, 21, (7)*, pages 739–755, 1991.

[122] Kaizhong Zhang. A constrained edit distance between unordered labeled trees. In *Algorithmica*, 1996.

[123] Kaizhong Zhang and Dennis Shasha. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM Journal of Computing, 18(6)*, pages 1245–1262, 1989.

[124] Kaizhong Zhang and Dennis Shasha. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms, 11(6)*, pages 581–621, 1990.

[125] Kaizhong Zhang, R. Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Proceedings Letters 42*, pages 133–139, 1992.

[126] Kaizhong Zhang, J. T. L. Wang, and Dennis Shasha. On the editing distance between undirected acyclic graphs and related problems. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407, 1995.