Titre: Etude comparative sur la détection de changements en XML

Auteurs: Grégory Cobéna
Talel Abdessalem
Yassine Hinnach

Contact : Grégory Cobéna

Gregory.Cobena@inria.fr
01 39 63 56 62

Adresse: INRIA - Projet Verso
Domaine de Voluceau - BP 105
78153 Le Chesnay Cedex
France

Thème: Bases de données semi-structurées

Mots-clefs: XML, Bases de données semi-structurées
Détection de changements, *diff*, Versions
Tree pattern matching

Résumé:

Dans le cadre de la gestion active de documents et bases de données, la détection de changements est un aspect essentiel de la gestion de versions. Le succès de XML a apporté un regain d'intérêt pour les algorithmes de diff sur des arbres et données semi-structurées. Récemment, plusieurs algorithmes et modèles ont été proposés, et nous avons souhaité mener une étude comparative de ces solutions. Pour cela, nous avons défini une grille d'analyse qui s'appuie sur un contexte d'application et les motivations correspondantes pour un diff. A partir de la, nous avons évalué sur le plan théorique et expérimental: (i) la qualité des résultats obtenus (ii) les performances de l'algorithme.

A comparative study for XML change detection

Grégory Cobéna Talel Abdessalem Yassine Hinnach

INRIA ENST ENST

April 24, 2002

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 (i) the quality of the results (ii) the performance of the tools. We also consider in which context each of these algorithms can be best used.

1 Introduction

The context for the present work is change control in XML data warehouses. There have been several proposals to represent changes on XML and semi-structured data [5, 20, 15], and build version management architectures [9].

In some applications (e.g. an XML 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 the 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.

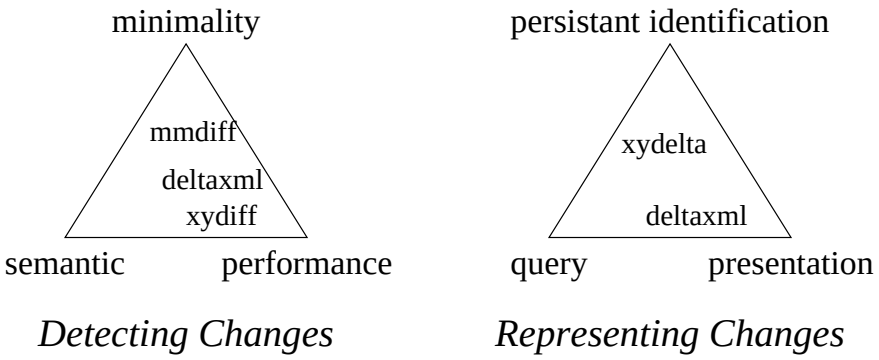

Figure 1: Trade-offs in change detection and representation

Observe that the *diff* we describe here are intended to XML documents. They 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 pieces of information such as the insertion of particular subtrees with a precise semantics, e.g. a new product in a catalog.

Several algorithms and representations of changes have been proposed recently. The goal of this survey is to analyze the different solutions based on motivations for using a *diff* tool in various applications. We study both the detection and the representation of changes.

The role of a *diff* algorithm consists in two parts: first it matches nodes between the two versions of the documents. Second it generates a document, namely a *delta*, representing a sequence of changes compatible with the matching.

While in some applications the performance (speed, memory) of the *diff* will be a key issue, in some others the focus will be on the minimality of the result (e.g. editing cost, file size) generated by the *diff*. In the world of XML, the semantic of data is also becoming extremely important and some applications may be looking for a semantically correct result, e.g. that a product in a catalog is identified by its name and that it is more likely that its price has been updated than its name. In Figure 1 we give a quick glance at the trade-off balance for some XML diffs.

Some of the proposed algorithms are able to detect *moves* between the two documents, whereas

others don't. We recall that most formulations of the change detection problem with *move* operations are NP-hard [34]. So the drawback of detecting *moves* is that such algorithms will only approximate the minimum edit script, whereas most algorithms on ordered trees provide a minimal editing script in quadratic time. The improvement when using *move* operation is that in some applications, users will consider that a *move* operation is less costly than a *delete* and *insert* of the subtree. In general, *move* operations are important to detect from a semantic viewpoint because they allow to trace nodes through time better than *delete* and *insert* operations. The semantic of *move* is to identify subtrees even when their context has changed. Some algorithms consider more semantics in XML documents. For instance, they may consider keys (e.g. ID attributes in the DTD) and match with priority two elements with the same tag if they have the same key.

Intuitively, the result of optimal algorithms (quadratic complexity) is slightly better when change rate is low (up to ten percent), and significantly better when the change rate is very high (more than thirty percent). However, algorithms supporting *move* operations can be very efficient when large subtrees have moved.

Different motivations will lead to different choices in the representation of changes as shown in Figure 1. First of all, changes represented in XML allow to support better quality services [24], in particular real query languages [29, 2], and they facilitate data integration. Unfortunately, [11] shows that there is no labeling of XML that is both good for querying and for persistent identification of nodes. In *XyDelta* [20], a delta contains a set of operations, and nodes are identified by ID numbers and positions. In *DeltaXML* [8, 15], editing operations are described inside the XML document with specific attributes. While both representations are formally equivalent, the first one serves more easily temporal query engines, whereas the second one makes it easier to present and integrate the results.

Our comparative study relies on experiments conducted over XML documents found on the web. Xyleme [31] crawled more than five hundred millions web pages and found four hundred thousand XML documents. Because only few of them changed during the time of the experiment (several months), our measures are based roughly on fifty thousand XML documents. Most experiments were run on only thirty thousand of them (because of the time it would take to run them on all the available

data). It would also be interesting to run it on private data (e.g. financial data, press data). Such data is typically more regular. We intend to conduct such an experiment in the future.

Finally, note that one of the authors participated in the development of the XyDiff program from INRIA. So our study may be biased by that. We did our best to avoid that bias.

In Section 2, we present the data model and recall briefly the motivations for studying XML diffs. The next section is an in-depth state of the art in which we present most algorithms in the area, and in particular the ones considered here. Then we give a summary of the tested programs in section 4. In the fifth section we present the data model and compare the two change representations. The next section is a speed and memory performance analysis. In Section 7, we study the quality of the results of diff programs. The last section concludes the paper.

2 Data Model and Motivations

In this section, we present the change data model and we consider how motivations for diff programs give us means to compare them.

2.1 Data Model

The data model we use for XML documents is based on their tree representation. Programs tested here work for ordered tree, although we also consider unordered-tree diff.

The change model is based on editing operations, namely *insert* , *delete* , *update* and *move* . These operations may be applied either to leaves or to subtrees. This captures the XML semantic better than tree operations previously defined in [27]. For instance, when a node is deleted, the entire subtree rooted at the node is deleted. Important aspects include management of positions in XML documents, and consistency of the sequence of operations depending on their order.

The *edit cost* of a sequence is defined by assigning a cost to each operation, usually 1 per node touched. The *edit distance* between document A and document B, is defined by the minimal editing cost over all sequences transforming A in B. A *delta* is minimal if its editing cost is no more than

the editing distance between the two documents. Each node in $A(B)$ that is not deleted(inserted) is matched to the corresponding node in $B(A)$. A *mapping* between two documents represents all matchings between nodes from the first and second documents. In some cases, a *delta* was said "minimal" if its editing cost is minimal for the restriction of editing sequences compatible with a given "mapping"[1].

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.

2.2 Motivations

As previously mentioned, an important motivation for change management in XML documents is the ability to query these changes, in particular when the *delta* is an XML document itself. The other aspect is version management, which means that the representation should allow for effective storage strategies and efficient reconstruction of versions of the documents. In both cases, support for indexing and persistent identification is useful. For instance, the prefix+postfix labeling of nodes allows to quickly compute ancestor/descendant tests and thus significantly improves querying. The persistent identification of nodes accelerates temporal queries and reduces the cost of updating an index. Work presented in [11] shows that there is no (short) labeling of XML that is good for both uses. As previously mentioned, the support of *move* operations is often important for temporal queries.

The other motivations concern mainly the detection of changes. One important aspect is that we suppose that all diffs are "correct", in that they find a set of changes that is sufficient to transform the old version into the new version of the XML document. In other words, they miss no changes.

For working on dynamic services and/or large amount of data, performance and low memory usage are mandatory. But minimality of the result is also important to save storage space and network bandwidth. Note that the storage efficiency and the effectiveness of version management are related both to the change detection and representation. We also mentioned the semantic aspect of XML data

[1] a sequence based on another mapping between nodes may have a lower editing cost

which becomes more and more important as shown in [28].

To illustrate, a real application that uses *XyDiff* is as follows: to permit concurrent work on large XML files and efficient updates of the modified parts, the diff between XML documents is computed. The semantic aspects allow to handle concurrent transactions by dividing the document into finer grain structures, which are identified in the DTD by ID attributes. The trade-off between performance and minimality then depends on the level of replication of the files in the network and on the relative cost of remote updates versus computing power.

3 State of the art

In this section, we present a rapid overview of the abundant previous work in this domain. Most XML and optimal tree pattern matching algorithms (e.g. MMDiff, DeltaXML) rely on the string edit problem. Other approaches (e.g. LaDiff, XyDiff) will first find a meaningful mapping between the two documents, and then generate a compatible representation of changes.

3.1 The String Edit Problem

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 [3, 17, 13, 1]. Insertion and deletion correspond to inserting and deleting a symbol in the string, each operation being associated with a cost. 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 prefix 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:

$$cost(delete(x[i])) + cost(x[1..i-1] \rightarrow y[1..j])$$
$$cost(insert(y[j])) + cost(x[1..i] \rightarrow y[1..j-1])$$
$$cost(subst(x[i], y[j])) + cost(x[1..i-1] \rightarrow y[1..j-1])$$

Note that for example $subst(x[i], y[j])$ is zero when the symbols are equals. This algorithm is often referred as the "Longest Common Subsequence", LCS.

The space and time complexity are $O(|x| * |y|)$. This algorithm has been improved by Masek and Paterson using the "four-russians" technique [21] in $O(|x|*|y|/log|x|)$ and $O(|x|*|y|*loglog|x|/log|x|)$ worst-case running time for finite and arbitrary alphabet sets respectively.

In [23], a $O(|x| * D)$ algorithm is exhibited, where D is the size of the minimum edit script. Such algorithms, namely *D-Band* algorithms, work when there is a small area of the matrix (close to the main diagonal) that contains all edit scripts of cost lower than D'. In this case, computation is first run on this smaller part of the matrix, and if an edit script if found of cost lower than D', then it must be the minimum edit script. In practical applications, most cost models have this property, and this technique is indeed used in the famous **GNU diff** [16] program.

3.2 Optimal Tree Pattern Matching

Serialized XML documents can be considered as strings, and we could use a LCS (Longest Common Subsequence) algorithm to detect changes. This can be useful for raw storage and raw version management, and can indeed be done using *GNU diff* that only supports flat text files. To support better services, it is preferable to consider specific algorithms for tree data.

Kuo-Chung Tai [27] gave a definition of the edit distance between ordered labeled tree and the first non-exponential algorithm to compute it. The time and space complexity is quasi-quadratic. Its semantic of operations is not well suited for XML data because it is similar to string editing, e.g. when a node is deleted, its children become children of the node's parent. 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. This model was presented in Selkow's variant [25], where the LCS algorithm described previously is used on trees in a recursive algorithm. Considering two documents $D1$ and $D2$, the time complexity is

$O(|D1| * |D2|)$. In the same spirit is Yang's [32] algorithm to find the syntactic differences between two programs.

In XML, we sometimes want to consider the tree as unordered. The general problem becomes NP-hard, but by constraining the possible mappings between the two documents, K. Zhang [33] proposed an algorithm in $O(|D1| * |D2| * (deg(D1) + deg(D2)) * log(deg(T1) + deg(T2))$, i.e. quasi quadratic. The mapping is constrained in that disjoined subtrees can only be matched to disjoined subtrees.

In [4], S. Chawathe presents an external memory algorithm, *XMDiff* (based on main memory version *MMDiff*), for ordered trees in the spirit of Selkow's variant. The CPU cost remains quadratic, and the memory usage is reduced but IO costs become quadratic. Intuitively, the algorithm constructs a matrix in the spirit of LCS, but some edges are removed to enforce that deleting(inserting) a node will delete(insert) the subtree rooted at this node. In this case, *D-Band* algorithm are difficult to use because of the weak connectivity in the matrix graph.

In a similar way, IBM developed a diff in [18] based on [12] and [26]. A first phase is added which consists in pruning identical subtrees based on their hash signature, but it is not clear if the result obtained is still minimal. Sun also released an XML specific tool named *DiffMK* [22] 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 tree structure of XML. Also the tests that we conducted, and other results found on the web seem to indicate that the current version is not "correct". In both cases, we had difficulties to use the available tools, and our technical comparison is limited. We were surprised by the relatively weak offer in the area of XML diff tools and we are not aware of more featured XML diff products from the top companies. We think that this may be due to a missing widely accepted XML change protocol. It may also be the case that some products are not publicly available. Fortunately, the quadratic algorithms of *MMDiff* and *XMDiff* are in spirit similar and their clear design represents well the spirit of today's tools.

One of the most featured product on the market, namely DeltaXML [14], uses a similar technique based on longest common subsequence computations. It uses Wu [30, 23] *D-Band* algorithm to run in linear time. Because the algorithm is applied at each level separately, the result is not strictly minimal.

The recent versions of DeltaXML supports the addition of keys -either in the DTD or as attributes- that can be used to enforce correct matching (e.g. always match a product by its name).

3.3 Tree pattern matching with a *move* operation

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 [34, 6] (by reduction from the "exact cover by three-sets"). One may want to convert in editing scripts a pair of *insert* and *delete* operations into a single *move* . The result obtained is in general not minimal, unless the cost of *move* operations is strictly identical to the total cost of *insert* and *delete* .

Recent work from S. Chawathe includes *LaDiff* [7, 6], 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. 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 when the change rate is maximized, the cost becomes quadratic in the size of the data.

Another algorithm has been proposed with one of the authors of the present paper *XyDiff* [10]. *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.

4 Summary of tested programs

In this section, we give a summary of the programs that we considered and we recall briefly their main characteristics. The list of tools is presented in Figure 2. We can group them into two sets.

The first set contains programs (e.g. MMDiff, XMDiff, Constrained Diff) that rely on a quadratic time algorithm to find the minimum edit script. In most case memory usage has been -or can be-

reduced to linear by using external memory (e.g. disk) as in [4]. The editing script found is not minimal if we consider *move* operations.

GNU diff applies a *D-Band* algorithm on the entire document and has a linear time and space complexity. We do not consider minimality because it does not support XML (or tree) editing operations. Note that in terms of simple text lines, the result is minimal.

DeltaXML's time complexity is $O(N * D)$ at each level of the document [30, 23], where N is the number of nodes, and D the edit distance between the two document levels. DeltaXML's result is not strictly minimal.

The other set (e.g. XyDiff, LaDiff) contains programs that first find a matching between the two documents, and then generate the (minimal) corresponding set of changes. The advantage of mapping-based programs is their support of *move* operations. These algorithms typically have a quasi-linear (e.g. $n * log(n)$) time and space complexity.

5 Comparison of the change representation models

In this section we recall the problematic of changes representation for XML documents, and we present the two recent proposals on the topic, namely *DeltaXML* and *XyDelta*.

While XML has been widely adopted both in academia and in industry, a standard is still missing for changes representation. Previous works [8] underline the necessity for querying semistructured temporal data. Recent works [8, 20, 15] propose data models to represent changes of XML documents with XML documents.

Previously, we presented the main motivations for representing changes as: querying changes, persistent identification of nodes, version management. For version management for instance, several strategies can be used: storing the only latest version of the document and all the deltas, or storing all versions of the documents, and computing deltas only when necessary. When only deltas are stored, their size must be reduced. The flat text representation of changes used by GNU diff is very efficient in terms of storage space and performance to reconstruct the documents. Its drawback are: (i) that it

Program Name	Author	Time	Memory	Moves	Minimal Editing Cost	Notes
fully tested						
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)
XyDiff	INRIA	linear	linear	yes	no	
not included in experiments						
LaDiff	Chawathe and al.	quadratic [2]	quadratic	yes	no	criteria based mapping
XMLTreeDiff	IBM	quadratic	quadratic	no	no	
DiffMK	Sun	quadratic	quadratic	no	no	no tree structure
XML Diff	Dommitt.com					we were not allowed to discuss it
Constrained Diff	K. Zhang	quadratic	quadratic	no	yes	for unordered trees constrained mapping

Figure 2: Quick Summary

is not XML and can not be used for queries (ii) files must be presented as flat text and this can not be used in native XML repositories.

DeltaXML: In [15] (or [8]), the delta information is stored in a "summary" of the original document by adding 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 document is not strictly validated by its DTD but has the same look and feel as the original document. There is some storage overhead when the change rate is low because: (i) position management requires to store the root of unchanged subtrees (ii) change status is propagated to ancestor nodes. It is also possible to store the whole document with changed data. A typical example would be:

```
<catalog deltaxml:modified>
  <product deltaxml:unchanged />
  <product deltaxml:modified>
    <status deltaxml:deleted>Not Available</status>
    <name>Digital Camera</name>
    <description>...</description>
    <price deltaxml:inserted>$399</price>
  </product>
</catalog>
```

XyDelta: In [20], every node in the original XML document is given a unique identifier, namely XID, according to some identification technique called *XidMap*. 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)[3]. The previous example would generate a following delta:

```
<xydelta
  v1_XidMap="(1-30)"
  v2_XidMap="(1-14;18-23;31-33;24-30)">
  <delete xid=(15-17) parent=6 position=1>
    <status>Not Available</status>
  </delete>
```

[3]the details for move and update operations can be found in [20]

```
  <insert xid=(31-33) parent=6 position=4>
    <price>$399</price>
  </insert>
</xydelta>
```

XyDeltas have nice mathematical properties, e.g. they can be aggregated, inverted and stored separately from the document. Also the persistent identifiers are useful in temporal applications. The drawback is that the delta does not contains contexts (e.g. ancestor nodes or siblings of nodes that changed) which is sometimes necessary to understand the meaning of changes or present query results. So the context has to be obtained, e.g. by processing the document.

There have been some other proposals, which in our opinion present serious drawbacks. For example, *Dommitt* delta for previous example would be:

```
<catalog>
  <product>
    <xmlDiffDeletestatus>
      <status>Not Available</status>
    </xmlDiffDeletestatus>
    <name>Digital Camera</name>
    <description>...</description>
    <xmlDiffInsertprice>
      <price>
        <xmlDiffInsertText>$399</xmlDiffInsertText>
      </price>
    </xmlDiffInsertprice>
  </product>
</catalog>
```

The problem is that for every type of document, a DTD has to be generated and it is different from the document's DTD.

5.1 Change Representation Experiments

Figure 3 shows the size of a DeltaXML delta and a XyDelta delta as function of the editing cost of the delta. Each dot represents the average[4] delta file size for deltas with a given editing cost. It confirms

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

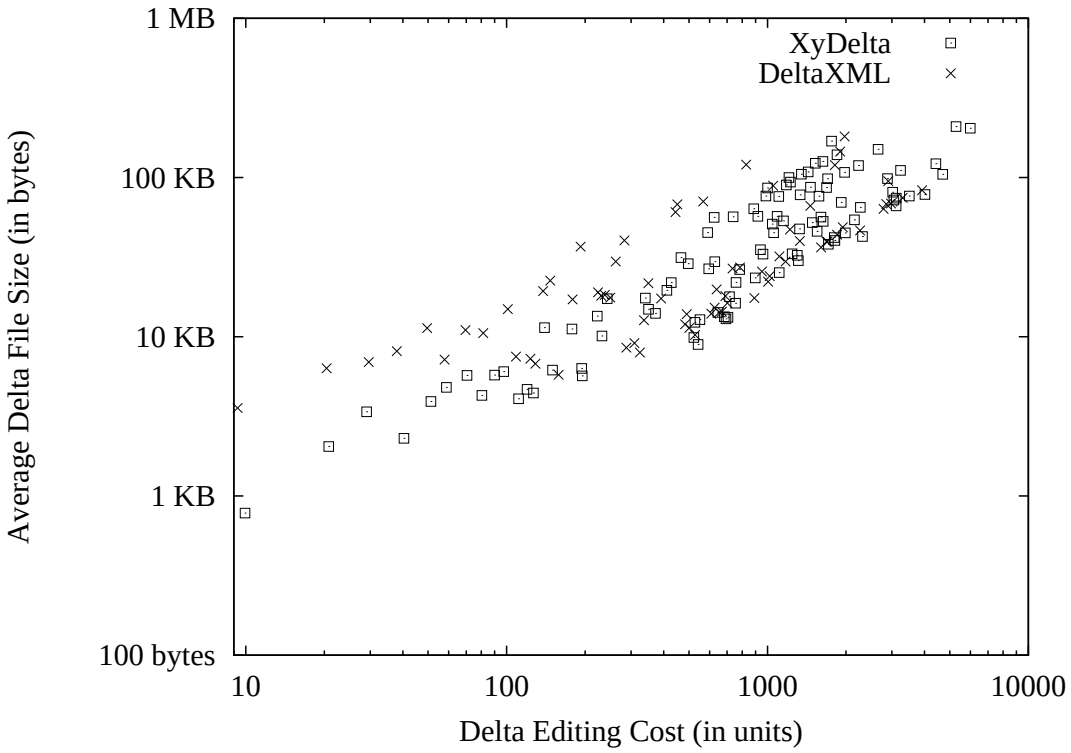

Figure 3: Size of the delta files

clearly that DeltaXML is slightly larger for lower editing costs because it describes many unchanged elements. On the other hand, when the editing cost becomes larger, its size is comparable to XyDelta. This figure is the result of more than ten thousand XML diffs, roughly twenty percent of the changing XML that we found on the web.

6 Speed and Memory usage

As previously mentionned, 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 XML data from DBLP [19].

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.

Let us analyze the behaviour of the time function plotted in Figure 4. On the one hand, *XyDiff* and *DeltaXML* are perfectly linear, as well as *GNU Diff*. On the other hand, *MMDiff* increase rate corresponds to a quadratic complexity. The computation times for *XMDiff* (the external-memory version of *MMDiff*) increases significantly when disk accesses become more and more intensive. Note that we used a 1Gb RAM PC to run *MMDiff* with enough memory for files up to hundred kilobytes.

In terms of implementation, *GNU Diff* is much faster than others because it doesn't parse or handle XML, while XML parsing represents for instance ninety percent of the time for *XyDiff*. This makes *GNU Diff* very suitable for simple version-based storage schemes. When handling large files, there are orders of magnitude between the running time of linear vs. quadratic algorithms.

Memory usage is also a key issue in that some algorithms do not scale to larger XML files. One can observe that with some work, this issue could be solved using, e.g., more economical space usage and secondary storage management techniques.

7 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 tools (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. Furthermore, we didn't consider here *move* operations. This was done by replacing each *move* operation by the corresponding pair of *insert* and *delete* .

In Figure 5, we present an histogram of the results, i.e. the number of documents in some range

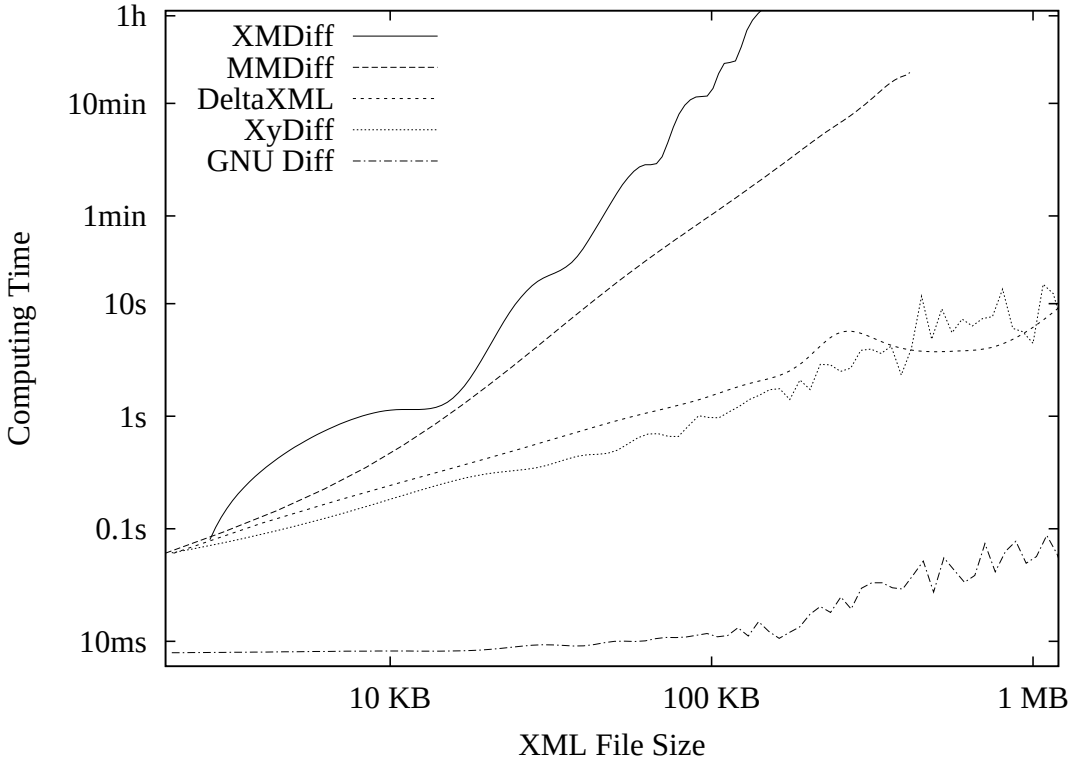

Figure 4: Speed of different programs

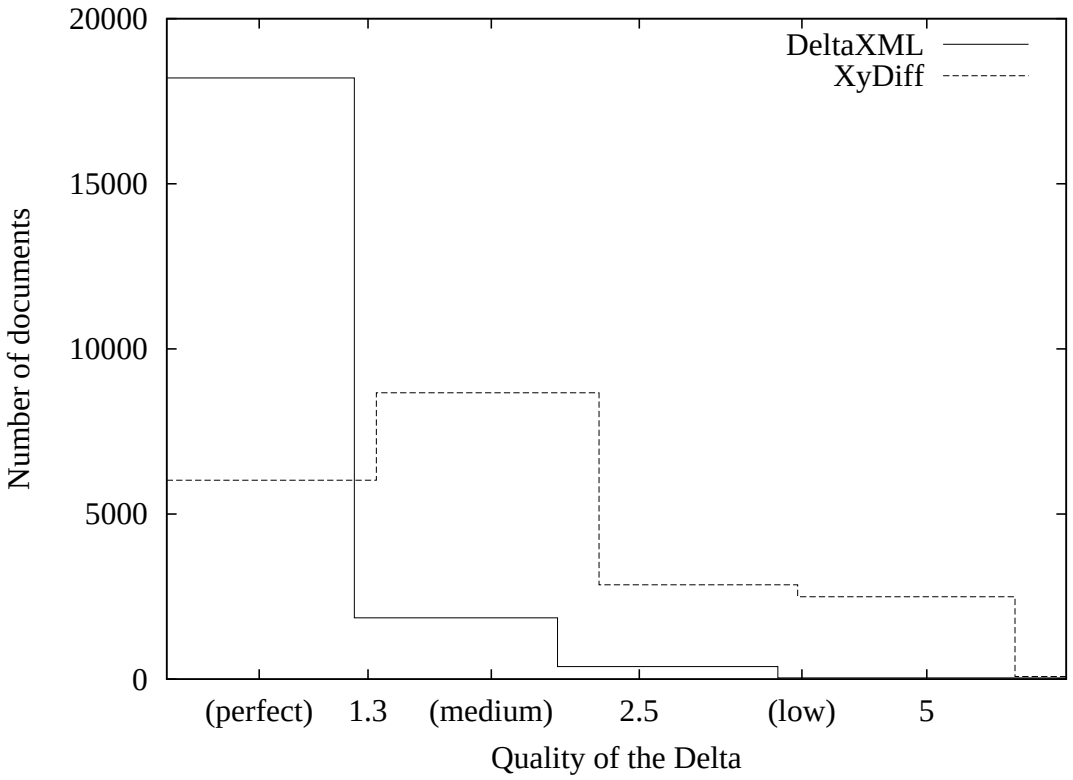

Figure 5: Quality Histogram

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.

These results in Figure 5 show that:

- **(i) DeltaXML:** For most of the documents, the quality of *DeltaXML* result is perfect (strictly equal to 1). For 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 cost on average twice the minimum.

In terms of file sizes, we also compared the different delta documents, as well as the flat text result of *GNU Diff*. They are on average similar.

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 a set of files is cheaper than copying them and deleting the original copy. On average, *XyDiff* performs a bit better, and it particular becomes better than *MMDiff* for five percent of the documents.

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* [7] which is a good example of semantic-based mapping and change detection.

8 Conclusion

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

We first presented a formal study of 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 experiments presented show (i) a significant quality advantage for minimal-based algorithms (DeltaXML, MMDiff) (ii) a dramatic performance improvement with linear complexity algorithms (DeltaXML, XyDiff, GNU Diff). DeltaXML [14] seems the best choice because it runs extremely fast and its results are close to the minimum. We also noted that flat text based version management (GNU Diff) still makes sense with XML for performance critical applications.

The second part of our study concerns the two recent proposals for change representation, we compare their features through analysis and experiments. Both support XML queries and version management, but the identification-based scheme (XyDelta) is slightly more compact for small deltas, whereas the delta-attributes based scheme (DeltaXML) is more easily integrated in applications. More work is clearly needed in that direction to define a common standard for representing changes.

Acknowledgments We would like to thank Serge Abiteboul, Vincent Aguilera, Robin La Fontaine, Amelie Marian, Tova Milo and Benjamin Nguyen for discussions on the topic.

References

[1] Wagner R. A. and M. J. Fischer. The string-to-string correction problem. *Jour. ACM 21*, pages 168–173, 1974.

[2] 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.

[3] A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*. Oxford University Press, 1997.

[4] S. Chawathe. Comparing hierarchical data in external memory. In *VLDB*, 1999.

[5] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *ICDE*, 1998.

[6] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD*, pages 26–37, Tuscon, Arizona, May 1997.

[7] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD*, 25(2):493–504, 1996.

[8] S. S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. *Theory and Practice of Object Systems*, 5(3):143–162, August 1999.

[9] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Version management of XML documents. In *WebDB (Informal Proceedings)*, 2000.

[10] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *ICDE*, 2002.

[11] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *PODS*, 2002.

[12] F.P. Curbera and D.A. Epstein. Fast difference and update of xml documents. In *XTech*, 1999.

[13] Sankoff D. and J. B. Kruskal. Time warps, string edits, and macromolecules. *Addison-Wesley, Reading, Mass.*, 1983.

[14] DeltaXML. Change control for XML in XML. www.deltaxml.com.

[15] R. La Fontaine. A delta format for XML: Identifying changes in XML and representing the changes in XML. In *XML Europe*, 2001.

[16] FSF. GNU diff. www.gnu.org/software/diffutils/diffutils.html.

[17] Levenshtein V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory 10*, pages 707–710, 1966.

[18] IBM. XML treediff. www.alphaworks.ibm.com/.

[19] Michael Ley. Dblp. dblp.uni-trier.de/.

[20] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. *VLDB*, 2001.

[21] W.J. Masek and M.S. Paterson. A faster algorithm for computing string edit distances. In *J. Comput. System Sci.*, 1980.

[22] Sun Microsystems. Making all the difference. http://www.sun.com/xml/developers/diffmk/.

[23] E. Myers. An o(nd) difference algorithm and its variations. In *Algorithmica*, 1986.

[24] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *SIGMOD*, 2001.

[25] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters, 6*, pages 184–186, 1977.

[26] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms, 11*, pages 581–621, 1990.

[27] K.C. Tai. The tree-to-tree correction problem. In *Journal of the ACM, 26(3)*, pages 422–433, july 1979.

[28] W3C. Resource description framework. www.w3.org/RDF.

[29] W3C. Xquery. www.w3.org/TR/xquery.

[30] S. Wu, U. Manber, and G. Myers. An o(np) sequence comparison algorithm. In *Information Processing Letters*, pages 317–323, 1990.

[31] Xyleme. `www.xyleme.com`.

[32] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience, 21, (7)*, pages 739–755, 1991.

[33] K. Zhang. A constrained edit distance between unordered labeled trees. In *Algorithmica*, 1996.

[34] K. Zhang, J. T. L. Wang, and D. 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.