Information Systems 78 (2018) 40–57

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

An efficient similarity-based approach for comparing XML documents

Alessandreia Oliveira[a,b,*], Gabriel Tessarolli[a], Gleiph Ghiotto[a,b], Bruno Pinto[a], Fernando Campello[a], Matheus Marques[b], Carlos Oliveira[b], Igor Rodrigues[a], Marcos Kalinowski[c], Uéverton Souza[a], Leonardo Murta[a], Vanessa Braganholo[a]

[a] Instituto de Computação, Universidade Federal Fluminense (UFF), Niterói, RJ, Brazil
[b] Departamento de Ciência da Computação, Universidade Federal de Juiz de Fora (UFJF), Juiz de Fora, MG, Brazil
[c] Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Rio de Janeiro, RJ, Brazil

ARTICLE INFO

Article history:
Received 14 October 2016
Revised 27 December 2017
Accepted 2 July 2018

Keywords:
XML
Diff
Match
Similarity

ABSTRACT

XML documents are widely used to interchange information among heterogeneous systems, ranging from office applications to scientific experiments. Independently of the domain, XML documents may evolve, so identifying and understanding the changes they undergo becomes crucial. Some syntactic diff approaches have been proposed to address this problem. They are mainly designed to compare revisions of XML documents using explicit IDs to match elements. However, elements in different revisions may not share IDs due to tool incompatibility or even divergent or missing schemas. In this paper, we present Phoenix, a similarity-based approach for comparing revisions of XML documents that does not rely on explicit IDs. Phoenix uses dynamic programming and optimization algorithms to compare different features (e.g., element name, content, attributes, and sub-elements) of XML documents and calculate the similarity degree between them. We compared Phoenix with X-Diff and XyDiff, two state-of-the-art XML diff algorithms. XyDiff was the fastest approach but failed in providing precise matching results. X-Diff presented higher efficacy in 30 of the 56 scenarios but was slow. Phoenix executed in a fraction of the running time required by X-Diff and achieved the best results in terms of efficacy in 26 of 56 tested scenarios. In our evaluations, Phoenix was by far the most efficient approach to match elements across revisions of the same XML document.

1. Introduction

The large use of XML documents has attracted great attention to this format. Multiple applications are required to export their data as XML documents, and several standards use XML as the basic structure to store and interchange data among systems. For instance, office applications such as MS Office store their data in a series of zipped XML documents; CASE tools store their UML models[1] in an XML document following the XMI[2] schema; IDEs store their metadata and scripts as XML documents; scientific experiments represent their workflow in XML. This popularity is a natural consequence of the simplicity and flexibility of XML.

XML users are often not only interested in the current data, but also in understanding their evolution over time [1]. As a result, multiple XML comparison approaches [2–9] were developed to deal with the specific idiosyncrasies of evolving XML documents. These approaches focus on identifying differences between two revisions of the same document, i.e., between versions resulting from the evolution of an XML document over time [10]. Therefore, they use techniques such as tree-to-tree correction [5,7], combined with node signature [9] or sub-tree signature [3], to identify the minimum set of modifications (i.e., inserts, deletes, and updates) that transforms one XML document revision into another. Hence, they were conceived for comparing document versions that belong to the same lineage (i.e., revisions).

Some existing approaches for comparing XML documents take advantage of the existence of primary keys in the compared documents. XyDiff [3], for instance, uses XML specific information such

* Corresponding author.
E-mail addresses: alessandreia.oliveira@ice.ufjf.br (A. Oliveira), gtessarolli@ic.uff.br (G. Tessarolli), ibarreto@ic.uff.br (G. Ghiotto), brunoferreirapinto@id.uff.br (B. Pinto), fernandocampello@id.uff.br (F. Campello), matheus.marques@ice.ufjf.br (M. Marques), carlosroberto@ice.ufjf.br (C. Oliveira), gmenezes@ic.uff.br (I. Rodrigues), kalinowski@inf.puc-rio.br (M. Kalinowski), usouza@ic.uff.br (U. Souza), leomurta@ic.uff.br (L. Murta), vanessa@ic.uff.br (V. Braganholo).

[1] UML - Unified Modeling Language (http://www.omg.org/spec/UML)
[2] XML - XML Metadata Interchange (http://www.omg.org/spec/XMI)

https://doi.org/10.1016/j.is.2018.07.001
0306-4379/

as an ID attribute to improve the matching among elements. The existence of an ID attribute for a given node provides a unique condition to match nodes: the nodes in both revisions must have the same ID value to be matched. In fact, the use of primary keys in XML has received attention in the past [11,12], and nowadays W3C provides a way to express them in XML Schema [13]. However, schemas are not mandatory and most XML documents do not have them [14–16]. Moreover, even when a primary key is present, there is no guarantee that it will remain intact during document evolution. For instance, UML models can be serialized through an XML schema named XMI, which employs IDs to identify elements. Although these IDs are consistent within a revision, CASE tools are not supposed to maintain the same IDs among revisions. Thus, IDs become unreliable for matching elements in this scenario.

In this paper we introduce Phoenix, an approach to compare XML documents using a recursive similarity computation to detect matching fragments among the document elements. Without relying on primary keys, our technique considers the similarity of four XML features (name, content, attributes, and sub-elements) using specific algorithms for computing the similarity of each feature. It then combines all these feature similarities into one overall element similarity. We then use an optimization algorithm to find matches of elements that maximizes the overall similarity among the documents. When the similarity of a match between two elements is below a specific threshold, Phoenix undoes the match and considers one element as an insertion (if it is present only in the newer version) and the other as a deletion (if it is present only in the older version). The same occurs for all other unmatched elements.

Matching elements using similarity is at the core of Phoenix, so we conducted two experimental evaluations using different versions of the Baltimore City Employee salaries dataset [17] to understand further its element matching sensitivity, efficacy, and efficiency. In the first evaluation, we conducted a sensitivity analysis to define the best similarity threshold for this domain. Therefore, we executed Phoenix several times, varying the similarity threshold it uses to determine whether two elements should be considered a match or not. This sensitivity analysis found 55% as the best threshold value for this domain, meaning that any match with similarity below 55% should be undone and interpreted as independent insertions and deletions. The second experimental evaluation analyzed the efficacy and efficiency of Phoenix, comparing it to two state-of-the-art XML diff approaches: X-Diff [9] and XyDiff [3]. We conclude that XyDiff is the fastest approach, but it fails in providing precise match results. Also, X-Diff achieved the most precise results in most of the comparisons (30 correct matches out of 56), but at the price of high execution times. Phoenix, on the other hand, was able to provide almost equivalent precision results (26 correct matches out of 56) within a fraction of X-Diff's execution time. Thus, in our evaluation, it was the most efficient approach by far.

The remainder of this paper is organized as follows. Section 2 presents a motivating example aiming at providing a better understanding of the problem and its relevance. Section 3 discusses related work. Section 4 describes the Phoenix approach and its characteristics. Section 5 details Phoenix's implementation. Section 6 describes the experimental evaluations and discusses the obtained results. Finally, Section 7 presents the conclusions and discusses future work.

2. Motivating example

This section presents a motivating example that will be used throughout the paper to illustrate the problem of comparing XML documents. Assume Gotham City provides information about its employees' salaries in XML format. The city administrators create a new document every year. Since crime and fraud occur all over the city, a worried citizen may want to analyze how this information evolved, comparing any two revisions of the XML document. With this comparison, she would be able to identify hired or fired/quitted/deceased employees, as well as changes in salary and job positions.

Assume the first version of this XML document (version 1) represents a legacy version that was inherited from the previous administration. Fig. 1 shows an excerpt of it, containing five employees. Each <*employee*> element stores the data of an employee, with attributes to represent the employee's name (unique in this dataset) and phone number, and sub-elements for representing the employee's job title, the agency's name and id she belongs to, when she was hired, her annual salary, and her gross pay. The XML document is generated from heterogeneous databases and, as a consequence, some attributes or elements may be missing. For instance, the "phonenumber" attribute of "Jim Gordon" is missing.

After trying to contact "Lee Thompkins", the new administration realized that her number was incorrect and noticed that the consistency of the data stored in the first version of this XML document was compromised. Consequently, the document was updated to a new version. For instance, among the changes, we can point out that the employee named *"Lee Thompkins"* had her annual salary and gross pay increased. While updating her annual salary, the person in charge noticed an error in the employee's name and phone number and fixed it. Thus, <*annualsalary*> changed to 60,050, <*grosspay*> changed to 52,588, <*agency*> was removed due to a normalization in the data, <*hiredate*> was added for the same reason, the *name* attribute changed to "Leslie Thompkins", and *phonenumber* changed to +1-424-121-6127. Fig. 2 shows the diff of versions 1 and 2. To improve the illustration, this figure uses different colors for some elements and attributes: red represents deletions from version 1, green represents additions in version 2, yellow represents updates and, finally, gray is used when there was no change. Considering this color schema, it is easy to see that an employee was hired (*"Lucius Fox"*) and another is missing (*"Kristin Kringle"* deceased) from version 1 to version 2. Also, all <*agency*> elements were deleted, and several elements changed.

As previously discussed, traditional approaches to compare XML documents are based on primary keys. In this example, the *name* attribute could be elected as a primary key to distinguish each employee from the others. However, when modifications such as changing the value of the elected key from Lee Thompkins to Leslie Thompkins take place, traditional approaches are usually not able to correctly match the elements in both versions.

3. Related work

The problem of comparing revisions of XML documents is not new. Existing approaches use techniques such as tree-to-tree correction, signature matching of sub-trees, or primary key matching to identify similar elements and then derive the diff. The diff (also called *delta* or *edit script*) is represented as a sequence of operations (*insert, delete*, and *update* – some also use *move*) that transforms one document version into the other. This delta can then be interpreted to find matching elements in both versions.

XyDiff [3] uses node signatures (a hash that is computed using the node value plus the signatures of the children of that node) to match sub-trees that did not change between versions. It works in a bottom-up fashion, matching leaf nodes first and propagating the matches to the subtrees above. Then it does a second top-down pass. Whenever there is more than one potential candidate for the matching, XyDiff uses a heuristic to pick one to avoid having to perform a full evaluation of the alternatives. It also uses XML specific information such as ID attributes whenever it is available. XyDiff models XML documents as ordered trees. Thus, besides con-