XML Data Integration Based on Content and Structure Similarity Using Keys

Waraporn Viyanon[1], Sanjay K. Madria[1], and Sourav S. Bhowmick[2]

[1] Department of Computer Science, Missouri University of Science and Technology
Rolla, Missouri, USA
[2] School of Computer Engineering, Nanyang Technological University, Singapore
wvz7b@mst.edu, madrias@mst.edu, assourav@ntu.edu.sg

Abstract. This paper proposes a technique for approximately matching XML data based on the content and structure by detecting the similarity of subtrees clustered semantically using *leaf-node parents*. The leaf-node parents are considered as a root of a subtree which is then recursively traversed bottom-up for matching. First, we take advantage of the "key" for matching subtrees which reduces the number of comparisons dramatically. Second, we measure the similarity degree based on data and structures of the two XML documents. The results show that our approach finds much more accurate matches with or without the presence of keys in XML subtrees. Other approaches experience problems with similarity matching thresholds as they either ignore semantic information available or have problems in handling complex XML data.

Keywords: XML, Similarity, keys, clustering.

1 Introduction

Data such as ACM SIGMOD Record [9] and DBLP [10] are published and shared using XML. However, these sources have similar contents but described using different tag names and structures. There are several methods [1, 4, 5, 6] for measuring XML content or structure similarity. They consist of extracting several features or keywords of each document and store them into an XML tree. Then, the similarity between XML documents is calculated by computing the edit distances between two trees which is a time-consuming task [1]. Similar arguments also hold for [4] which uses Brute-Force algorithm for comparing path similarity degree. On the other hand, some approaches like LAX [5] and SLAX [6] are based on XML document's characteristics in terms of its depth and number of instances contained to cluster the document into subtrees which are used to calculate the similarity. Though they outperform edit distance based-schemes, they ignore semantic information available such as "keys"; rather they rely on finding subtrees or "*clustering point*" which does not work for all XML data.

The concept of key [2] has been introduced for XML documents. In this paper, we propose a new approach called XML Document Integration or XdoI in short which considers both the data structure and the content for approximately matching XML documents to integrate XML data sources together. More specifically, our approach detects the similarity of two subtrees clustered semantically from two XML documents taking advantage of XML keys for subtree matching in bottom-up fashion. This reduces the number of comparisons dramatically. After we match the subtrees based on keys, the remaining unmatched subtrees are processed for finding similarities in both its content and structure to select the best matched subtrees. We outperform LAX and SLAX schemes in terms of false positives during the best matching subtrees as well as in terms of result quality and execution time of finding similarity matches using keys.

2 Related Work

David Buttler [3] summarized three approaches on structure similarity: (1) Tag similarity, (2) Tree Edit Distance (TED), and (3) Fourier Transform Similarity. Tag similarity algorithm is not satisfactory in terms of accuracy as the pages conforming to the same schema, such as HTML, have only a limited number of different tags; one page may contain a large number of a particular tag, while the comparison page may contain relatively few occurrences of the tag. In Fourier Transform Similarity [7], the basic idea is to remove all the information from a document except for its start and end tags, leaving a skeleton that represents the structure. This algorithm is proved in [3] to be the least accurate of any of approximation algorithm, while also being the slowest. Path similarity [3] measures are expensive to compute as there are n! mappings between the paths of two trees.

3 Motivation and Problem Statement

LAX approach [5]divides XML trees into subtrees by considering a clustering point from the height (distance from the furthest child) and the number of link branches of XML trees. A *link branch* is a link between two *candidate elements* which have at least two children or the distance to its furthest child is at least three. The subtree can be generated by deleting the link branch below the clustering point. The *clustering point* is calculated from the *maximum weighting factor* of the multiplication between the height level and the number of link branches (ex. tinyDB (2,3) has weight of 6 in Figure 1) .

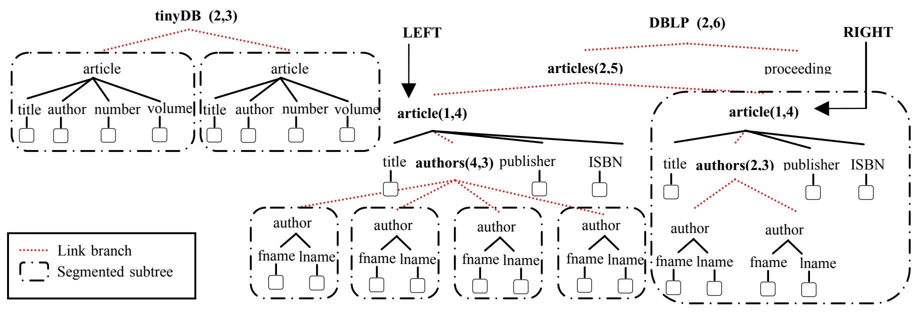

Fig. 1. Example of LAX clustering on two different XML structures: DBLP tree has two article elements, part A is the left article and part B is the right article

Figure 1 shows LAX clustering on two different XML documents with dissimilar structures. The clustering points on the tinyDB tree are the link branches under the candidate element tinyDB(2,3) because of the maximum weight = 2*3 = 6. Therefore, the tinyDB tree is clustered into two subtrees rooted by the article nodes but the DBLP tree is clustered into four subtrees rooted by the author nodes for the first article since the maximum weight is from the element authors(1,3). It shows clearly that the resulting subtrees in both documents are different in terms of semantics. The first tree is cut into a set of articles but the second tree in Figure 1 (part A) is chopped into a set of authors. The results of clustering affect next steps such as subtree matching because it compares different types of objects. In addition, the leaf-nodes "title", "publisher" and "ISBN" of the first article on the second tree are not considered in any subtree.

Consider the second article on the second XML tree in Figure 1 (part B). The whole article is clustered into one subtree since the maximum weight is 2*5 = 10 from the element article(2,5). This example shows that we may get different kinds of subtrees in the XML document and therefore, similarity scores will vary. Missing elements such as "title", "publisher" and "ISBN" from the first article also occur after the clustering.

The authors of [6] found that when LAX is applied after fragmenting documents, the hit subtrees selected from the output pair of fragment documents that have large tree similarity degrees might not be the proper subtrees to be integrated. SLAX [6] is therefore an improved LAX to solve this problem. SLAX divides XML documents into smaller portions by parsing XML documents into K document trees. In each document tree, SLAX applies the weighting factor from LAX to find points for subtree clustering. This way they try to solve the issue of matching right subtrees but the problem with LAX clustering discussed above still occurs in divided trees. We elaborate this problem further in Section 5.

In this paper, we show how to address the drawbacks of the clustering discussed above by using leaf-node parents as a clustering point using bottom-up approach while exploiting keys and recursively matching subtrees bottom-up.

4 Our Approach

XDoI approach is split into three phases for simplicity and understanding. In Phase I, we cluster the base XML tree and target XML tree by considering leaf-node parents as clustering points. The clustered subtrees in the base XML tree are considered independent items that will be matched with clustered subtrees in the target XML tree in Phase II. The best matched subtrees are integrated in the first XML tree as a resulting XML tree in Phase III.

4.1 Basic Definitions

Definition 1: XML Document Tree - An XML document tree T is an ordered labeled tree parsed from an XML document. Let T_b and T_t be two XML document trees, where b denotes the base tree and t denotes the target tree. The T_b and T_t are clustered into subtrees later.

Definition 2: Leaf-node parent - A leaf-node parent is a node that has at least a child as a leaf node. This node is considered as a root of a subtree in the clustering process.

Definition 3: Clustering point - The clustering point is the link above the leaf-node parent. The clustering point indicates the place for clustering an XML tree into subtree(s).

Definition 4: Simple subtree – A simple subtree is a clustered tree with only a root and leaf nodes.

Definition 5: Complex subtree – A complex subtree is a clustered subtree with at least one simple subtree, a root and one or more of leaf nodes.

4.2 Preprocessing

We do pre-processing on the XML documents by parsing and storing XML documents in relational tables based on their tree structure using XRel [8]. Storing XML data in a relational model for similarity findings addresses the issues of scalability which affects all other schemes. XRel uses four relations (Element, Attribute, Text and Path) to store the data and structure of the XML documents and a document relation to store the complete XML document as shown in Table 1.

In this paper, we find a leaf-node value match for all unique node values with the same path using the SQL statement shown in Table 2. The unique leaf node is considered as the key which can identify the subtree. It is possible that XML documents may not contain a key in some subtrees or an item in either base or target XML tree has a key but the same item may not appear in the other XML tree. So, we need to continue to find the best matching for this case by comparing rest of the subtrees.

Table 1. XRel relational schema and subtree table

Element(docid, pathid, start, end, index, re-index)
Attribute(docid, pathid, start, end, value)
Text(docid, pathid, start, end, value)
Path(pathid, pathexp)
Document(docid, document)
Subtree(docid, subtreeid, pathid, start, end, key, value)

Table 2. Finding key(s)

```
SELECT docid, pathid, value
FROM subtree
GROUP BY docid,
PathID,Value
HAVING Count(Value) = 1
```

4.3 Phase I: Clustering XML Tree into Subtrees

An XML tree can be parsed into small independent items by clustering it into more meaningful subtrees. Each clustered subtree represents independent items. A well-clustered subtree requires as in [5] (1) each subtree to represent one independent item, (2) each independent item is clustered into one subtree and (3) the leaf nodes belonging to that item should be included in the subtree.

For our approach, the way of clustering an XML tree into an independent item is to use leaf-node parents as clustering points discussed in Definitions 2 and 3. The leaf-node parents are considered as a root of each subtree. The clustered subtrees are categorized into two types, *simple subtree* and *complex subtree*, defined in Definitions 4 and 5. They are stored in the subtree table shown in Table 1 and will be used in subtree matching later.

4.4 Phase II: Matching Subtrees

The keys in the base subtrees found in the pre-processing step are used to match with the keys in the target subtrees. Matching the keys of base XML tree and target XML tree will reduce the number of unnecessary subtree matching. It is possible that one to multiple matching occurs in this step. The number of one to multiple matching will be reduced after finding the similarity degree using Definitions 6.1 & 6.2. The non-matched subtrees will be also needed to find subtree similarity degree for each pair.

To find the correct matched subtree, we consider both the content and the structure of the base and target XML trees by comparing PCDATA value and signatures to decide which subtrees are the right matched pair. First, we define the **Subtree Similarity Degree (SSD)** of each subtree in the definition below.

Definition 6.1: Subtree Similarity Degree (measure1) -- Let t_{bi} be the base subtree and t_{tj} be the target subtree. Assume n is the number of leaf nodes having the same PCDATA value. Let n_{bi} represents the number of leaf nodes in t_{bi} and n_{tj} represents the number of leaf node in t_{tj}. For score rule each common node is assigned 1 point and a common node defined as a key is assigned 2.

$$s_1(t_{bi}, t_{tj}) = \frac{n}{n_{bi}} \times 100\% \tag{1}$$

Definition 6.2: Subtree Similarity Degree (measure2) – This is the ratio of common matched leafnode values between the base and target subtrees.

$$s_2(t_{bi}, t_{tj}) = \frac{2 \times n}{n_{bi} + n_{tj}} \times 100\% \tag{2}$$

SSD (measure2) is used to eliminate the number of one to multiple matches in case of having the maximum SSD (measure1) with overlap target subtrees.

Definition 7: Matched Subtree - The matched pair of subtrees t_{bi} and t_{tj} is the pair that has the *maximum subtree similarity degree* from Definitions 6.1 and 6.2. The maximum subtree similarity degree is considered as a matched subtree.

$$SM[i] = Max(s_1(t_{bi}, t_{tj})) \tag{3}$$

Definition 8: Path Similarity Degree (PSD) -- PSD is a best matched subtree by comparing the number of nodes in the base path on the matched leaves that have the same labels excluding leaf nodes. This is applied in case of having the same maximum similarity degree with target subtrees from Definition 7.

$$PSD(i) = \frac{N}{N_{bi}} \times 100\% \tag{4}$$

N denotes the number of nodes in the base path that have the same labels with those in the target path, N_{bi} denotes the total number of nodes in the base path, and *i* is the total number of matched leaf node paths in the base subtree between 1 to *k* paths.

Definition 9: Path Subtree Similarity Degree (PSSD) -- After PSD calculation in Definition 8, the PSSD is determined by the following equation.

$$PSSD(t_{bi}, t_{tj}) = \frac{\sum_{i=1}^{k} PSD(i)}{k} \times s_1(t_{bi}, t_{tj}) 100\% \tag{5}$$

4.5 Phase III: Join Algorithm

Let S_b and S_t be two XML data sources and each XML document $d_b \in S_b$ and $d_t \in S_t$ be clustered into XML document trees T_b and T_t. T_b and T_t are clustered into subtrees, t_{bi} and t_{tj} where subscript i denotes number of subtrees in the base document tree and j denotes number of subtrees in the target document tree. The steps in joining two XML documents consist of (1) Finding the similarity degree of each pair of subtrees. If there are more than one matched subtrees, find the maximum similarity degree among them, (2) Calculate the tree similarity degree from mean value of similarity degrees of matched subtrees and (3) If the tree similarity degree is greater than a given threshold τ, where $0 \le \tau \le 1$, the two documents can be integrated at the clustering point.

4.6 Algorithm

Our approach can be written in a pseudocode as follows. This algorithm is processed after XML documents are parsed into a relational database. There are four main modules:

```
Algorithm XDoI: Input: XML documents $d_b$ and $d_t$ and Output: Set of matched subtrees pairs ($t_{bi}$, $t_{tj}$)
    //Module1 Identifying key(s)
        Define key();                                   //*** Table 2: Finding key(s) ***
    //Module 2: Clustering the XML trees
        Find_leafnode_parent();                         //***Sub Module
        ClusterXMLTree();                               //***Sub Module - Using leaf-node parent
    //Module 3: Matching subtrees
    Match_with_key();                                   //***Table 3 (Query 3)***
    // Subtree Similarity degree computation
    for (every $t_{bi}$ in $d_b$) {                     //Subtrees from the based document
        MaxSim[i] = 0;
        for ($t_{tj}$ in $d_t$){
            CalSimilarity S($t_{bi}$ ,$t_{tj}$)          //***Definition 6.1&2 ***
            MaxSim[i] = Max(MaxSim[i], S($t_{bi}$ ,$t_{tj}$));   //***Definition 7 ***
        }
        StoreMSSD($t_{bi}$, $t_{tj}$, MaxSim[i]);       //Store MSSD in a temp table
    }                                                   // Path Subtree Similarity degree computation
      for (every $t_{bi}$ in MSSD, such that Count MaxSim() >1 and MaxSim > $\tau$ ) {
        MaxPath[i] = 0                                  //Match subtree more than one pair
        for (j = 1 to $k_t$) {
            CalPathSimilarity P($t_{bi}$ ,$t_{tj}$)      //***Definition 8 ***
            MaxPath[i] = Max(MaxPath[i], P($t_{bi}$ ,$t_{tj}$));   //***Definition 9 ***
        }
        StorePSSD($t_{bi}$, $t_{tj}$, MaxPath[i]);      //Store MSSD in a temp table
    }
    //Module4: Integration
    for (every $t_{bi}$ in PSSD, such that MaxPath > $\tau$ ){   // similarity degree > the threshold
        $d_i$ = integrate($S_b$, $S_t$)                 //*** Section 4.5 ***
        return ($d_i$);
      }
    }
Sub Modules:
Find_leafnode_parent(){
    for every $p_i$ from the PATH table
    {      if lastpathsection($p_i$) is not attribute {
            lnp = Remove the last path section from ($p_i$);
            store_lnp(lnp);                             //store a leafnode parent into a temporary table
           }
     }
ClusterXMLTree()  {
    for (i in all_lnp){
        $region_i$ = Retrieve leafnode parent info      //***Table 3 (Query 1)***
        $t_i$ = find_subtree($region_i$);               //***Table 3 (Query 2)***
        store_subtree($t_i$)
    }
  }
```

Fig. 2. XDoI Algorithm

Table 3. SQL—Subtree clustering and match with key

Query1: Retrieve leafnode parent information: **Select** distinct e.docid, e.pathid, e.st, e.ed **From** tmp_leafnode_parent p , element e **Where** p.docid = e.docid and p.ppathid = e.pathid
Query2: Find subtrees: **Select** docid, pathid, st, ed, ++subtreeid, value **From** txt **Where** st >= region.st and ed <= region.ed
Query 3: Match subtrees with keys: **Select** s1.subtreeid, s2.subtreeid **From** subtree s1, subtree s2 **Where** s1.docid = 1 and s2.docid = 2 and s1.key = 'Y' and s2.key = 'Y' and s1.value = s2.value

(1) Preprocessing, (2) Clustering, (3) Matching subtrees and (4) Integrating matched subtrees. In matching subtree phase, we first match subtrees with keys according to Query 3 in Table 3. The first unmatched subtree in the first XML document is compared with all the subtrees in the second XML document and this procedure is recursively done for all the subtrees in the first XML document. The best match can be calculated by following the algorithm steps in Figure 2.

5 Performance Evaluation

In this section we conduct experiments to observe the efficiency and effectiveness of our algorithm comparing with algorithms LAX and SLAX [5, 6]. We used Intel Pentium 4 CPU 2.80GHz with 1GB of RAM running on Window XP Professional with Sun JDK 1.6.0_02 and Oracle Database 10g Standard Edition. We used available bibliography datasets, SIGMOD Record [9], 482 KB as the base document and three segmented documents of DBLP.xml [10], 700 KB each as the target documents. We also used some synthetic XML datasets classified according to keys, types of structure (shallow and deep) and file sizes.

5.1 Experimental Results

First we evaluate the variation of clustered subtrees among the algorithms, XDoI, LAX and SLAX. We compare the clustering points (subtree roots) and the number of subtrees using SigmodRecord and DBLP. Table 4 shows the difference of the clustered subtree from the three approaches. XDoI clustered subtrees by leafnode parents which cover all leafnode values on XML documents. This guarantees that the associated values are not missed while clustering. The clustering of LAX and SLAX is done according to the weighting factor *w* discussed in Section 3. The clustered subtrees from SLAX rely on *K* value mentioned in Section 3. From the result of clustering SigmodRecord, we got different segmented subtrees at three different levels "issue", "article" and "author" which can be integrated with different kind of bibliography documents. On the other hand, LAX got only "issue" level which affects the similarity scores when we compared the LAX's clustered subtrees with clustered subtrees rooted at "article" level from the DBLP dataset. SLAX clusters XML documents depending on the selected *K* value, however, the right *K* value required for clustering across different XML documents is not clear [6] as no obvious procedure was provided to select the appropriate value of *K*. This also causes the loss of some meaningful information such as "issue volume" and "issue number" when $K > 4$ for

Table 4. Clustered point and number of subtrees from different approaches

XML document	XDoI		LAX		SLAX		
	subtree-root element	Number of subtrees	subtree-root element	Number of subtrees	*K*	subtree-root element	Number of subtrees
SigmodRecord	issue	67	Issue	67	k≤4	issue	67
	article	1504			k>4	article	1504
	authors	1504					
DBLP(dblp01.xml)	inproceedings	769	Inproceedings	769	Any k	inproceedings	769

SigmodRecord. Similarly, *K* has different values for different documents as shown in Table 4. Therefore, the LAX's and SLAX's clustering can affect the results of subtree similarity degree substantially and therefore, can integrate mismatched subtrees.

Next we evaluate the execution time of clustering subtrees, finding key(s), computing the subtree similarity degrees and the path subtree similarity degrees of three pairs of SigmodRecord and three selected fragments of DBLP. We call these pairs as 1st pair, 2nd pair and 3rd pair in our experiments. We ran the experiments using $\tau = 0.5$ as a user-defined "threshold value". We used the three pairs of a dataset from SigmodRecord and DBLP which do not have key(s) defined so that the finding keys module is required. On the other hand if we integrate two XML documents with predefined keys such as XML documents generated from RDBMS, the finding keys module is not necessary. In case of hybrid XML documents with only some portions in the documents having key(s) identified and some parts do not, the execution time for the finding keys module will take less time for documents with non-predefined keys. In the experiments, our approach though requires two more modules, finding keys and mapping with key, the execution time in total is less than SLAX's. This is because SLAX clusters the XML documents into subtrees using the weighting factor so most of the execution time is used to calculate the factor and the key mapping module in XDoI reduces the number of subtrees needed to calculate SSD and PSSD.

In the experiments, we also compared our approach using keys with non-keys to observe the margin of improvements using keys in terms of the overall execution time. The numbers in parentheses in Table 5 are execution time of XDoI with non-keys. Obviously it takes more time than the one with keys because it has to calculate SSD for each subtree pair. The comparison for each module against other approaches is shown in Figure 3.

Table 5. Execution time (in seconds) for clustering & key generation in SigmodRecord and DBLP Data

Module	XdoI			SLAX		
	1st pair	2nd pair	3rd pair	1st pair	2nd pair	3rd pair
Clustering	17625	20070	22344	233482	281576	258624
Finding keys	156811	154796	184983	-	-	-
Mapping with keys	217358	209748	89484	-	-	-
SSD	15975258 (37149128)	17697580 (45629554)	53201074 (59464448)	17775897	20445114	36746179
PSSD	162200 (185655)	32063 (40594)	890605 (1082337)	962885	935901	1385309
Total Time	16529252 (37352408)	18114257 (45690218)	54388490 (60569129)	18972264	21662591	38390112

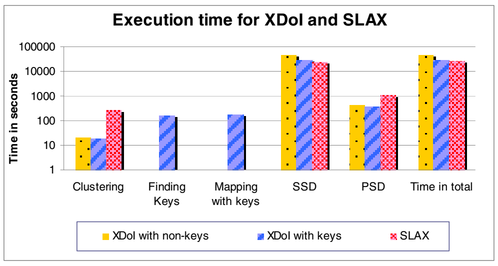

Fig. 3. Average execution time for XDoI and SLAX

Table 6. Matched subtrees of SigmodRecord and DBLP

Threshold	**XDoI with keys**			**SLAX**		
	1st pair	**2nd pair**	**3rd pair**	**1st pair**	**2nd pair**	**3rd pair**
50%	361(339,22) FP = 6.09%	336(322,14) FP = 4.17%	93(67,26) FP=27.96%	314(273, 41) FP=13.06%	286(225,61) FP= 21.33%	102(55,47) FP=46.08%

We tested the effectiveness of the two approaches by measuring the false positives in the number of matched pairs. Table 6 shows the number of matched subtrees from our experiments and the values in parentheses indicate the number of correctly matched subtrees and the number of incorrectly matched subtrees respectively. From the results, we observe that all the incorrect matched pairs are simple subtrees rooted by the "authors" element overlapped with a complex subtree rooted by the "article" element. They are matched with a target subtree because the subtrees have the same authors in both base and target subtrees but they are from different articles. The false positive rates from the experiments show that our approach with keys has a much lower false positives compared to SLAX. Therefore, our approach can precisely detect the proper matched subtrees for integrating XML documents than SLAX.

5.2 Result Quality

In this section, we observe and compare the result quality of the similarity degrees generated by our approach and SLAX for different types of XML files. We define the result quality of subtree matching as $Q = Sn/An$, where Sn is the number of matched subtrees by a given approach and An is the number of actual matched subtrees. Figures 4 (a) and (b) show the result quality for XML documents of file sizes ranging from 1KB to 71KB. The results show that our approach and SLAX have similar performance for shallow and semi-shallow XML documents but for large and deep XML documents XDoI has much better result quality compared to SLAX. The performance of SLAX drops from 100% to 0% since it does not cluster the complex XML documents into proper subtrees like the problem discussed in Section 3.

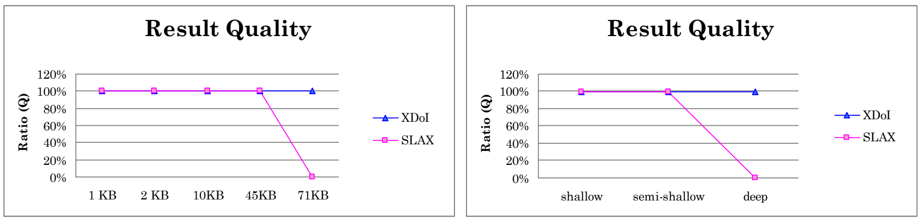

Fig. 4. (a) and (b): Result quality for different file sizes and file types

6 Conclusions

In this paper, we presented a data-centric approach to cluster XML documents into subtrees by using leaf-node parents and keys to reduce the number of subtrees matching and improve the similarity degree by reducing false positives. From performance evaluation, we have shown that the keys are very efficient in finding more appropriate subtrees matching among XML documents. Our approach performs better than SLAX and LAX for complex XML documents as we address the drawbacks of these schemes using “keys” and an efficient bottom-up matching algorithm.

References

1. Bille, P.: Tree Edit Distance, Alignment Distance and Inclusion, ISBN 87-7949-032-8
2. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.: Keys for XML. Computer Networks 39(5), 473–487 (2002)
3. Buttler, D.: A Short Survey of Document Structure Similarity Algorithms. In: International Conference on Internet Computing 2004, pp. 3–9 (2004)
4. Liang, W., Yokota, H.: A Path-sequence Based Discrimination for Subtree Matching in Approximate XML Joins. In: Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW 2006), pp. 23–28. IEEE, Los Alamitos (2006)
5. Liang, W., Yokota, H.: LAX: An Efficient Approximate XML Join Based on Clustered Leaf Nodes for XML Data Integration. In: Jackson, M., Nelson, D., Stirk, S. (eds.) BNCOD 2005. LNCS, vol. 3567, pp. 82–97. Springer, Heidelberg (2005)
6. Liang, W., Yokota, H.: SLAX: An Improved Leaf-Clustering Based Approximate XML Join Algorithm for Integrating XML Data at Subtree Classes. In: Proceedings of DBWeb 2005, IPSJ Symposium Series (16), pp. 41–48 (2005)
7. Rafiei, D.: Fourier-Transform Based Techniques in Efficient Retrieval of Similar Time Sequences. Thesis of University of Toronto (1999)
8. Yoshikawa, M., Amagasa, T.: XRel: A Path-based Approach to Storage and Retrieval of XML Documents. In: Proceedings of the 19th IEEE International Conference of Data Engineering (ICDE), India, pp. 519–530 (2003)
9. ACM SIGMOD Record in XML, `http://www.acm.org/sigmod/record/xml`
10. XML Version of DBLP, `http://dblp.uni-trier.de/xml/`