XANDY: A Scalable Change Detection Technique for Ordered XML Documents Using Relational Databases

Erwin Leonardi Sourav S. Bhowmick

School of Computer Engineering, Nanyang Technological University, Singapore 639798

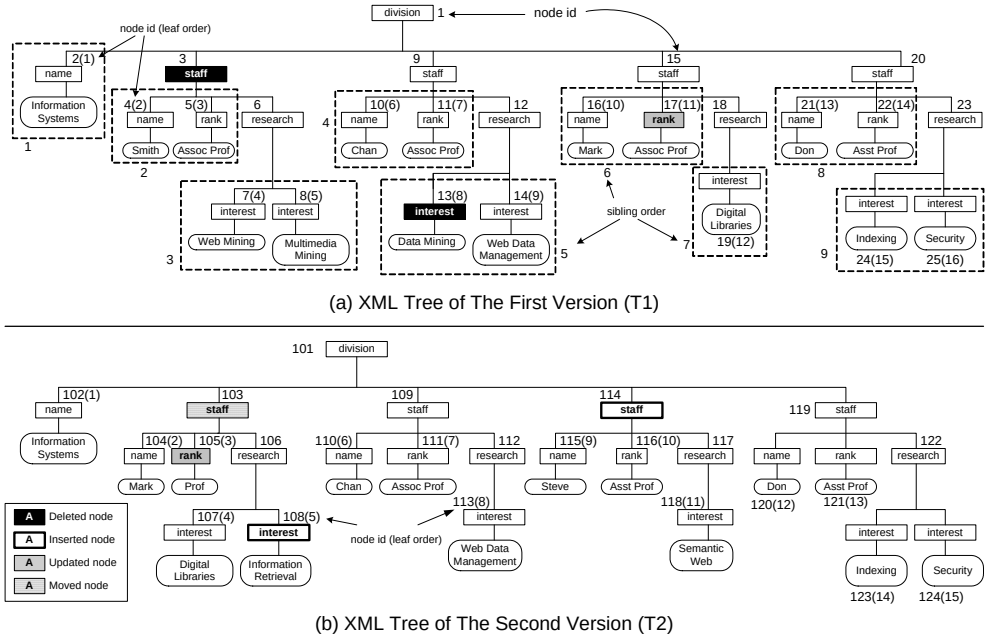
Abstract

Previous work in change detection to XML documents is not suitable for detecting the changes to large XML documents as it requires a lot of memory to keep the two versions of XML documents in the memory. In this article, we take a more conservative yet novel approach of using traditional relational database engines for detecting the changes to large *ordered* XML documents. To this end, we have implemented a prototype system called XANDY that converts XML documents into relational tuples and detects the changes from these tuples by using SQL queries. Our experimental results show that the relational-based approach has better scalability compared to published algorithm like X-Diff. It has comparable efficiency and result quality compared to X-Diff in some cases. Our experimental results also show that, generally, XANDY has better result quality than XyDiff.

1 Introduction

Over the next few years XML is likely to replace HTML as the standard format for publishing and transporting documents over the Web. The Web allows these documents to change at any time and in any way. These changes typically take two general forms. The first is existence. XML pages exhibit varied longevity pattern. The second is structure and content modification. An XML document replaces its antecedents, usually leaving no trace of the previous document. These rapid and often unpredictable changes to the information create a new problem of detecting and representing these changes (hereafter called *XML deltas* or *XDeltas*). Such a change detection tool is important to incremental query evaluation, trigger condition evaluation, search engine, data mining applications, and mobile applications [4,15].

Even though the underlying challenge is how to detect and represent the



(a) XML Tree of The First Version (T1)

(b) XML Tree of The Second Version (T2)

Fig. 1. Tree Representation of Ordered XML Documents.

changes to large volume of data, the novel context of the XML forces us to significantly extend traditional techniques. XML data is commonly modelled by a tree structure (hereafter called *XML tree*), where nodes represent elements, attributes and text data, and parent-child pairs represent nesting between XML elements. The XML trees are classified into *ordered* trees and *unordered* trees. An *ordered* tree is one in which both the ancestor relationships and the left-to-right ordering among siblings are significant. An *unordered* tree is one in which only ancestor relationships are significant. In this article, we focus on *ordered* XML documents.

The changes to ordered XML documents can be classified into two types: *changes to internal elements* and *changes to leaf elements*. An *internal element* does not contain textual data. For example, consider two versions of an XML document in Figure 1. For the time being, ignore the dotted boxes. The nodes 3 and 9 in Figure 1(a) are internal elements. The changes to internal elements are called *structural changes* as they modify the structure of the document but do not change the textual data content. We consider the following types of *structural changes*: *internal element insertion*, *internal element deletion*, and *internal element movement*. For instance, node 114 in Figure 1(b) is an example of internal element insertion. Node 15 (node 103 in Figure 1(b)) is moved from being the fourth child of node 1 to be the second child of node 101. Note that node 101 in T_2 is the corresponding node of node 1 in T_1. A *leaf element* is an element/attribute which contains textual data. For example, node 4 is a leaf element which has name "name" and textual content "Smith". The changes to leaf elements are called *content changes* as they modify the textual data content. We consider the following four types of *content changes*: *leaf element insertion*, *leaf element deletion*, *content update of a leaf element*,

and *leaf element movement*. For example, a leaf element "Interest" (id=108) which has value "Information Retrieval" is an inserted leaf element. In this article, we present novel techniques for detecting the *content* and *structural* changes in *ordered* XML documents using relational databases.

1.1 Related Work

The XML change detection problem is related to the problem of change detection to trees. In [2], the authors address the problem of detecting changes to two snapshots of hierarchically structured information that are represented as *ordered* trees. MH-Diff [1] is an efficient algorithm for meaningful change detection between two *unordered* trees. The authors introduce the following matching criteria to compare nodes, and the matchings between two versions of a tree are determined based on this assumption.

Given two labeled trees, T_1 and T_2, there is a "good" matching function, so that given any leaf s in T_1, there is at most one leaf in T_2 that is "close" enough to match s.

The faster version of the matching algorithm uses longest common subsequence computations for every element node starting from the leaves of the document. The algorithm runs in time $O(ne+e^2)$, where n is the total number of leaf nodes, and e is a *weighted edit distance* between the two trees. This assumption holds well for many SGML documents that do not contain duplicate or similar objects, but it does not hold for many XML documents.

Recently, a number of techniques for detecting the changes to XML data has been proposed. Most of these techniques focus on developing main memory algorithm to detect the changes. XMLTreeDiff [5] and XyDiff [4] are designed for detecting the changes to ordered XML documents. In XyDiff, the changes are detected by using signatures and weights of nodes. For each node in a XML DOM tree, the signature is computed using the nodes content and its children signatures. Simultaneously, the weight is computed for each node, based on the size of its content for text nodes and the sum of the weights of its children for element nodes. The change detection starts from finding a matching between the heaviest nodes. Note that the heavier subtree will have higher priority to be chosen for comparison. Once a match is found, it is propagated to the ancestors and descendants nodes to get more matchings. Inserts, deletes and moves are computed after all exact matches are found. XMLTreeDiff (a tool developed by IBM) is a set of JavaBeans and does ordered tree-to-tree comparison to detect the changes to XML documents by using DOMHash [10]. X-Diff [15] is designed for computing the XDeltas for two unordered XML documents. The main strength of X-Diff algorithm is that it reduces the mapping space significantly and helps the algorithm to achieve polynomial time in complexity. However, the change detection response time is slower than XyDiff. XMLTreeDiff [5], XyDiff [4], and X-Diff [15] are the

memory-based approaches as they parse both versions of XML documents and detect the changes to these documents that are in the main memory.

The above memory-based approaches have some limitations as follows. First, they require the entire trees (i.e., DOM trees) of two XML documents to be memory resident. This problem is exacerbated by the fact that these trees are typically much larger than their XML documents [9]. Thus, the scheme is not scalable for very large XML documents. In fact, the scheme is inefficient. We need to parse an XML document whenever we want to compare it with a new version. That is, if a document is compared with more than one document at different times, then it has to be parsed multiple times.

1.2 Motivation

There has been a substantial research effort in storing and processing XML data. The relational storage approach has attracted considerable interest with a view to leveraging their powerful and reliable data management services. The above limitations coupled with the recent success in storing XML data in relational databases [6,7,12,13,17] force us to ask whether we can address these problems by using relational techniques to detect the changes to XML documents. A relational database can be used in two ways to address the change detection problem. Let us elaborate on this further. Suppose source A sends a XML document D_1 (version 1) at time t_1 to source B. B stores D_1 in its local RDBMS. At time t_2, A modifies D_1 to D_2 (version 2) and sends it to B. B can now detect the changes to the document in the following two ways.

(1) B extracts D_1 from the relational database and compares it to D_2 (before inserting D_2 into the database) by using any one of the above memory-based change detection approaches.
(2) B first stores D_2 in the relational database and then detects the changes to the documents by executing a set of SQL queries whenever appropriate.

In the first approach, the costs incurred are the extraction time of D_1 and the change detection time of the memory-based algorithms. However, as mentioned earlier, these algorithms are not scalable. Furthermore, the extraction cost is incurred every time we wish to compare D_1. The costs incurred by the second approach are the time taken to insert D_2 into the database and the change detection time in the database. In particular, by storing XML documents as tables, we can filter out tuples and attributes that are not needed. Second, the system using this approach is more scalable as it can handle very large XML documents that may not fit into the main memory. Third, by storing XML in RDBMS, we only need to parse the XML documents once and then we can find the changes by issuing SQL queries against the database. Finally, implementing a change detection algorithm in SQL makes the programming task easier. Also, as SQL is an industry standard and available on

all major RDBMS, the implementation of the change detection technique is portable.

As the relational storage approach for storing and managing XML data has gained popularity, we believe that the second approach is an attractive option if it can address the following two issues. First, the insertion and extraction times for D_1 and D_2 should be comparable. In other words, the underlying relational storage structure must support efficient insertion and extraction of XML documents. Second, we must be able to detect all types of changes accurately.

1.3 Our Approach

In our preliminary efforts in [3,8], we have demonstrated that it is indeed possible to use the relational database to detect the changes to *ordered* XML documents. However, the approaches in [3,8] focused on the content changes only and did not detect the structural changes. The underlying relational schema of DiffXML [3] is simplistic and is not efficient for path expressions query processing. Hence, our approach in [8] uses `SUCXENT` schema that enables us to insert, extract, and query XML data efficiently [12].

In this article, we present a novel relational-based approach called Xandy (**X**ml en**A**bled cha**N**ge **D**etection s**Y**stem) for detecting the both content and structural changes to *ordered* XML documents. Given T_1 and T_2 as the old and new versions of an XML document respectively, first, we store both documents in the relational database. After the documents are stored in the relational database, we are ready to detect the changes between T_1 and T_2. There are two phases in our approach to detect the changes between T_1 and T_2 as follows:

(1) **Find the Best Matching Subtrees**.
The objective of this phase is to find the most similar subtrees in T_1 and T_2. In this phase, we try to match the subtrees in T_1 to ones in T_2. Some of the subtrees in T_1 can be matched to more than one subtree in T_2 and vice versa. We measure the similarity of each matching subtrees by calculating the *similarity score* of these matching subtrees. The most similar subtrees are called *best matching subtrees*. The *top-down approach* starts computing the similarity scores from the root nodes of T_1 and T_2, and move downward. In the *bottom-up approach*, we start matching the root nodes of the subtrees rooted at the lowest level, and move upward. We shall see that the bottom-up approach is, on average, 5 times faster then the top-down approach. We also shall see that the result quality of the bottom-up approach is better than the one of the top-down approach.

(2) **Detect the Changes**.
In this phase, we use the information on best matching subtrees in order to detect the types of changes as discussed above by issuing SQL queries. First, we determine the changes on internal nodes (both insertions and

deletions). Next, the inserted and deleted leaf nodes are detected. Finally, we detect updated leaf nodes and moved nodes. The XDeltas are stored in the relational tables.

1.4 Contributions

In summary, this article makes the following contributions:

- We propose a novel technique to detect the changes, both *structural* and *content* changes, to the *ordered* XML documents by using relational databases. The relational-based approach is able to overcome the scalability problem that occurs on the memory-based approach.
- By extending a published relational schema called SUCXENT [12], XANDY is efficient not only for detecting the changes, but also for inserting, extracting, and querying XML data as it inherits the features of SUCXENT. In [12], the authors have shown that the execution time of insertion and extraction XML documents by using `SUCXENT` schema are comparable.
- An extensive performance study was conducted on our approaches. The experimental results show that the relational-based approach is more scalable than the memory-based approaches.

The organization of the rest of this article is as follows. In Section 2 we shall briefly discuss the relational schema that we use for storing the XML documents. In Section 3, we discuss how we are able to find the best matching subtrees from two given versions of an XML document. We present the algorithms for the *top-down approach* and the *bottom-up approach*. We shall elaborate how the XDeltas can be discovered in Section 4. We also present the SQL queries that are used to discover the XDeltas. In Section 5, we compare the performance of different approaches. Finally, we conclude the article in the last section.

2 Background

There are two approaches for storing XML documents in relational database: the *model-mapping* approaches [6,7,12,17] and the *structure-mapping* approaches [13]. The *model-mapping* approaches maintain a fixed schema which is used to store XML documents irrespective of their schemas. The *structure-mapping* approaches first create a relational schema based on the schemas of XML documents. In this article, we also adopt the model-mapping approach due to the following reasons. First, the DTD or any structure definition for the documents may not be available. Second, it is often the case that an XML collection would have documents that conform to more than one DTD. Detecting the changes to these XML documents would be a problem in the *structure-mapping* approach as several relational schemas would be created, one for each DTD. This means that we have to rewrite a different set of SQL queries for detecting the

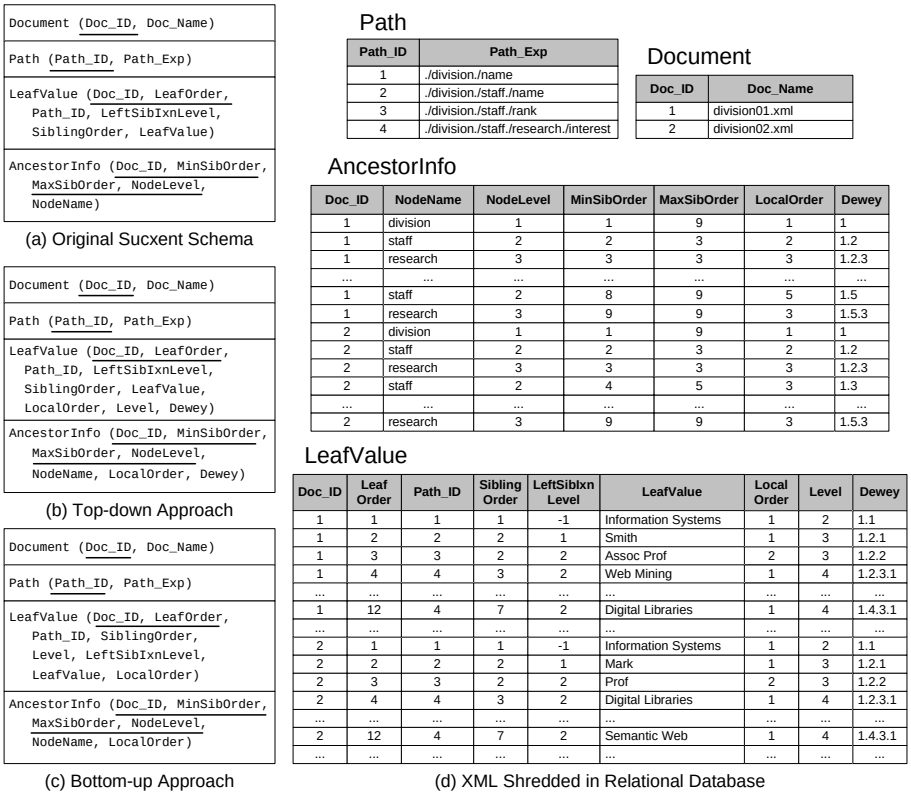


Fig. 2. The SUCXENT Schemas and XML data in RDBMS.

changes to XML documents which have different DTD. Third, the DTD of an XML document may also be changed. Consequently, the relational schema of the underlying database has to be modified. Obviously, this will result in modification of the SQL queries for detecting the changes.

We have extended the relational schema of our XML storage system called `SUCXENT` (**S**chema **U**n**C**oncious **X**ML **EN**abled Sys**T**em) [12]. We chose `SUCXENT` because we have shown in [12] that our approach outperforms significantly the current state-of-the-art model mapping approaches like XParent [7] as far as storage size (up to 20%), insertion time (up to six times), extraction time, and path expression queries (up to 25%) are concerned. Note that Jiang et al. has shown in [7] that XParent outperforms existing model mapping approaches such as Edge approach [6], and XRel [17]. The `SUCXENT` schema is shown in Figure 2(a). The `Document` table is used for storing the names of the documents in the database. This allows us to store multiple versions of XML documents. The `Path` table is used to record all paths from the root to the leaf nodes. It maintains the path ids and the relative path expressions as instances of the `Path_ID` and `Path_Exp` attributes respectively.

The `LeafValue` table is used for storing the information on leaf nodes. The `Doc_ID` attribute indicates which XML document a particular leaf node belongs to. The `Path_ID` attribute maintains the id of the path of a particular

leaf node stored in the `Path` table. The `LeafOrder` attribute is used to record the *node order* of the leaf nodes in an XML tree. For example, consider the XML tree in Figure 1(a). When we parse the XML document, we will find the leaf node "name" with value "Information Systems" as the first leaf node in the document. Hence, we assign the `LeafOrder` equal to "1" for this leaf node. The next leaf node we find is the node "name" with value "Smith". Therefore, the `LeafOrder` of this node is equal to "2". Two leaf nodes have the same `SiblingOrder` if they share the same parent. For example, the leaf nodes with `LeafOrder` equal to "2", and "3" shall have the same `SiblingOrder` (equal to "2") since they share the same parent node (node "staff" with node id 3). The dotted boxes in Figure 1(a) indicate the leaf nodes that have the same `SiblingOrder`. The `LeftSibIxnLevel` (Left Sibling Intersection Level) is a level at which the leaf nodes belonging to a particular sibling order intersect the leaf nodes belonging to the sibling order that comes immediately before. For example, consider the leaf nodes with `SiblingOrder` equal to "3" in the XML tree. These leaf nodes shall intersect with the leaf nodes having `SiblingOrder` equal to "2" at the node "staff" (id=3) which is at level 2. The `LeafValue` stores the textual content of the leaf nodes. Note that the `LeftSibIxnLevel` in this table is only useful for constructing the XML documents from the relational database [12].

The `AncestorInfo` table is used for storing the ancestor information for each leaf node. The `Doc_ID` attribute indicates to which XML document a particular ancestor node belongs to. We record the names and the level of ancestor nodes in the `NodeName` and `NodeLevel` attributes respectively. The `MinSibOrder` and `MaxSibOrder` store the minimum and maximum sibling orders of the leaf nodes under a particular ancestor node respectively. For example, the node "staff" (id=3) in Figure 1(a) has `MinSibOrder` and `MaxSibOrder` equal to "2" and "3" respectively. Node "division" (id=1) has `MinSibOrder` and `MaxSibOrder` equal to "1" and "9" respectively.

For the top-down approach, the `SUCXENT` schema is modified as follows. The attributes `LocalOrder` and `Dewey` are added in the `LeafValue` and `AncestorInfo` tables to store the position of a node among its siblings and ancestors' local orders of each node respectively. This `DEWEY` attribute is adopted from the Dewey Ordering Encoding [11]. For example, the Dewey values of nodes 3 and 7 in T_1 are "1.2" and "1.2.3.1" respectively. The local order is assigned in an incremental manner among the siblings from left to right. We also add the attribute `Level` in the `LeafValue` table to store the level of leaf nodes. The extended `SUCXENT` schema for the top-down approach is shown in Figure 2(b). Figure 2(d) depicts the relations containing two shredded XML documents in Figure 1 (partial view only).

We extend the `SUCXENT` schema for the bottom-up approach as follows. We add the `Level` and `LocalOrder` attributes in the `LeafValue` table to store

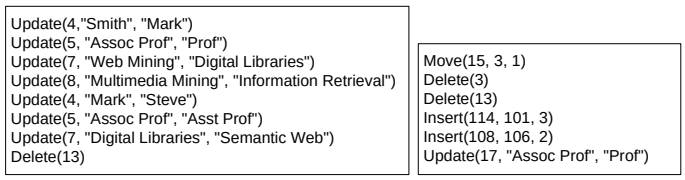


Fig. 3. XDeltas: Example.

the level and the position among siblings of the leaf nodes respectively. The `AncestorInfo` table is extended by adding the `LocalOrder` attribute that is used to store the positions among siblings of the internal nodes. We do not use the Dewey attribute in this approach for the following reason. Our approach determines the best matching subtrees at level $level + 1$ before finding the best matching subtrees at level $level$. That is, the matching subtrees at level greater than $level$ are already determined. Hence, we just need to use the information on the best matching subtrees at level $level + 1$ in order to find the best matching subtrees at level $level$. The extended `SUCXENT` schema for the bottom-up approach is depicted in Figure 2(c). Figure 2(d) depicts the relations containing two shredded XML documents in Figure 1 (partial view only, without the `Dewey` attribute in the `LeafValue` and `AncestorInfo` tables).

Note that the performance of the extended `SUCXENT` schema is comparable to the performance of the original `SUCXENT` schema and still outperforms XParent. As the modifications are not significant, the insertion and extraction performances of the extended `SUCXENT` schema are still faster than XParent. The modified `SUCXENT` still stores the ancestor information of only the leaf nodes compared to XParent which stores ancestor information of every node. Hence, the storage requirement of the extended `SUCXENT` schema is still lesser than the one of XParent. The query processing performance of the extended `SUCXENT` schema is also faster than XParent as the key properties of the original `SUCXENT` schema (`SiblingOrder`, `MinSibOrder`, `MaxSibOrder`, etc.) are still preserved in the extended schema. Furthermore, as query processing in modified `SUCXENT` is still done without θ-joins, query performance is still better than XParent due to the reduced storage space.

3 Finding Best Matching Subtrees

In this section, we shall elaborate how to find the best matching subtrees. The objectives of finding the best matching subtrees are to enable us to get the *minimum XML delta*. The *minimum XML delta* can be defined as the delta which has the least number of edit operations (types of changes).

Suppose we have two XML trees, T_1 and T_2, as depicted in Figure 1. There are more than one XDelta that can be detected from T_1 and T_2. For example, we may have an XDelta that contains seven updates and a deletion as shown in Figure 3(a). We get this XDelta if we match subtree t_3 to subtree t_{103}, subtree

t_9 to subtree t_{109}, subtree t_{15} to subtree t_{114}, and subtree t_{20} to subtree t_{119}. We can also have other XDelta as depicted in Figure 3(b). This XDelta that contains six edit operations is a result of matching subtree t_9 to subtree t_{109}, subtree t_{15} to subtree t_{103}, and subtree t_{20} to subtree t_{119}. The second XDelta is a candidate to be the *minimum XML delta* if there is no other XDeltas that have lesser number of edit operations. Therefore, the selection of the correct matching subtrees is important in order to get the minimum XML deltas.

3.1 Preliminaries

The matching subtrees from the first and second versions of an ordered XML tree are determined by measuring their similarity. The *similarity score* is used to measure the degree of similarity between two subtrees in the two versions of an XML document. Note that a subtree in the first version may be matched to more than one subtree in the second version. The most similar subtrees are considered as the *best matching subtrees*. In this section, we shall introduce some concepts that we shall be using to compute *similarity score*. First, we present the notations that will be used in our discussion as follows.

- $L(T)$: a set of leaf nodes in the subtree T,
- $I(T)$: a set of internal nodes in the subtree T.

The root node of subtree T is denoted by $root(T)$. The textual content of a leaf node ℓ_x is denoted by $value(\ell_x)$, where $\ell_x \in L(T)$. The name and level of node n are denoted by $name(n)$ and $level(n)$ respectively.

Definition 3.1 [**Matching Leaf Nodes**] *Let $\ell_1 \in L(T_1)$ and $\ell_2 \in L(T_2)$ be two leaf nodes from the first and second versions of an XML documents respectively. ℓ_1 and ℓ_2 are **matching leaf nodes** (denoted as $\ell_1 \leftrightarrow \ell_2$) if $name(\ell_1) = name(\ell_2)$, $level(\ell_1) = level(\ell_2)$, and $value(\ell_1) = value(\ell_2)$, where $\ell_1 \in L(T_1)$ and $\ell_2 \in L(T_2)$.* □

Example 3.1 The leaf nodes 2 and 102 are matching leaf nodes ($\ell_2 \leftrightarrow \ell_{102}$) because they have the same node name ("name") and the same node value ("Information Systems"). Note that a leaf node in T_1 can be matched to more than one leaf node in T_2, and vice versa. The leaf node 111 in T_2 can be matched to the leaf nodes 5, 11, and 17 in T_1 as they have the same node name ("rank") and the same node value ("Assoc Prof").

The matching leaf nodes are classified into two types: *fixed matching leaf nodes* and *shifted matching leaf nodes*. The *fixed matching leaf nodes* are the ones whose positions among their siblings are not changed. The *shifted matching leaf nodes* are the ones whose positions among their siblings are changed due to the insertions or deletions of their siblings, and changes of their positions among theirs siblings. For example, nodes 2 and 102 are fixed matching leaf nodes, and nodes 14 and 113 are shifted matching leaf nodes. Note that if ℓ_1

and ℓ_2 are not matching leaf nodes, then they are denoted by $\ell_1 \not\leftrightarrow \ell_2$.

Next, we define the notion of *matching sibling orders* that will only be used in the bottom-up approach. A set of leaf nodes that have the same parent node will have the same sibling order. The matching sibling orders can be seen as the summarization of the matching leaf nodes. Consider subtrees t_9 and t_{109} as depicted in Figure 1. There are three matching leaf nodes in subtrees t_9 and t_{109} ($\ell_{10} \leftrightarrow \ell_{110}$, $\ell_{11} \leftrightarrow \ell_{111}$, and $\ell_{14} \leftrightarrow \ell_{113}$). We are able to summarize these matching leaf nodes to two matching sibling orders. Hence, the storage space needed for storing the matching information is reduced.

Definition 3.2 [**Matching Sibling Orders**] *Let so_1 and so_2 be two sibling orders in T_1 and T_2 respectively. Let $siborder(\ell)$ be the sibling order of a leaf node ℓ. Let $P = \{p_1, p_2, ..., p_x\}$ and $Q = \{q_1, q_2, ..., q_y\}$ be two sets of leaf nodes in T_1 and T_2 respectively, where $\forall p_i \in P$ $siborder(p_i) = so_1$, $\forall q_j \in Q$ $siborder(q_j) = so_2$, $P \subseteq L(T_1)$, and $Q \subseteq L(T_2)$. Then so_1 and so_2 are the* ***matching sibling orders*** *(denoted by $so_1 \Leftrightarrow so_2$) if $\exists p_i \; \exists q_j$ such that $p_i \leftrightarrow q_j$ where $p_i \in P$ and $q_j \in Q$.* □

Example 3.2 A set of leaf nodes whose parent is node 9 in T_1 has a sibling order equal to 4. A set of leaf nodes whose parent is node 109 in T_2 has a sibling order equal to 4. These two sibling orders are matching sibling orders as they have two matching leaf nodes ($\ell_{10} \leftrightarrow \ell_{110}$ and $\ell_{11} \leftrightarrow \ell_{111}$).

The next step is to determine the *possible matching subtrees*. Informally, the *possible matching subtrees* are subtrees in which they have at least one matching leaf node. Hence, the subtrees in T_1 are possible to be matched to more than one subtree in T_2. From these possible matching subtrees, we determine the most similar subtrees to be the best matching subtrees. Note that the matching is only performed between subtrees at the same level. This is because matching the subtrees at different level is an expensive process. Formally, the *possible matching subtrees* is defined as follows.

Definition 3.3 [**Possible Matching Subtrees**] *Let t_1 and t_2 be two subtrees rooted at nodes $i_1 \in I(T_1)$ and $i_2 \in I(T_2)$ respectively. t_1 and t_2 are the* ***possible matching subtrees*** *(denoted by $t_1 \simeq t_2$) if $name(i_1) = name(i_2)$, $level(i_1) = level(i_2)$, and $\exists p \; \exists q$ such that $p \leftrightarrow q$, where i_1 is the ancestor of p, i_2 is the ancestor of q, $p \in L(T_1)$, and $q \in L(T_2)$.* □

Definition 3.3 is a general definition of the possible matching subtrees for both the top-down and bottom-up approaches. We are able to use the notion of *matching sibling orders* in the third condition of Definition 3.3 for the bottom-up approach as there is at least one matching leaf node in the matching sibling orders (Definition 3.2).

Example 3.3 The subtrees rooted at node 9 in T_1 and node 109 in T_2 are possible matching subtrees ($t_9 \simeq t_{109}$) as they have three matching leaf nodes

```
Input:
  document id of the first version did1,
  document id of the second version did2,
  threshold theta
Output:
  the MATCHING table

1   root1 = getRootNode(did1)
2   root2 = getRootNode(did2)
3   if (root1.name != root2.name) then
4      return
5   end if
6   score = calculateScore(root1, root2)
7   if (score == 1.0) then
8      Store the root node information
                  in the MATCHING table
9      return
10  else if (score < theta) then
11     return
12  end if
13  Store the root node information
                    in the MATCHING table
14  TD-findBestMatchingSubtree(did1, did2,
                    theta, root1, root2);
15  extendIdenticalSubtrees(did1, did2);
```

(a) The *rootMatching* Algorithm

```
1  SELECT
2    M.DID1, M.DID2, A1.MINSIBORDER,
3    A1.MAXSIBORDER, A2.MINSIBORDER,
4    A2.MAXSIBORDER, A1.NODELEVEL,
5    A1.LOCALORDER, A2.LOCALORDER,
6    1.000 AS SCORE
7  FROM MATCHING AS M,
8       ANCESTORINFO AS A1,
9       ANCESTORINFO AS A2
10 WHERE
11   M.DID1 = did1 AND
12   M.DID2 = did2 AND
13   M.SCORE = 1.000 AND
14   A1.DOC_ID = did1 AND
15   A2.DOC_ID = did2 AND
16   A1.NODENAME = A2.NODENAME AND
17   A1.NODELEVEL = A2.NODELEVEL AND
18   A1.LOCALORDER = A2.LOCALORDER AND
19   A1.MINSIBORDER >= M.MINSO1 AND
20   A1.MAXSIBORDER <= M.MAXSO1 AND
21   A2.MINSIBORDER >= M.MINSO2 AND
22   A2.MAXSIBORDER <= M.MAXSO2
```

(b) Extend Identical Subtrees

Fig. 4. Top-down Approach: Algorithm *rootMatching* and SQL Queries.

($\ell_{10} \leftrightarrow \ell_{110}$, $\ell_{11} \leftrightarrow \ell_{111}$, and $\ell_{14} \leftrightarrow \ell_{113}$). The subtrees rooted at node 15 in T_1 and node 109 in T_2 are also possible matching subtrees ($t_{15} \doteq t_{109}$) as they have one matching leaf node ($\ell_{17} \leftrightarrow \ell_{111}$).

3.2 Best Matching Subtrees

The next step is to determine the *best matching subtrees* from a set of possible matching subtrees. Note that the terms matching subtrees and matching internal nodes are used interchangeably. Consequently, we have to measure how similar two possible matching subtrees are. Note that two subtrees are more similar if they have more numbers of matching leaf nodes. We are able to use the proportion of the matching leaf nodes and the total nodes in the subtrees for determining how similar two subtrees are. We define a metric called *similarity score* to measure how similar these subtrees are.

Definition 3.4 [**Similarity Score**] *The similarity score* $\Re$ *of two subtrees* t_1 *and* t_2 *is as follows:* $\Re(t_1, t_2) = \frac{2|A|+|B|}{|t_1|+|t_2|}$ *where* $|t_1|$ *and* $|t_2|$ *are the total numbers of leaf nodes in* t_1 *and* t_2 *respectively, and* $|A|$ *and* $|B|$ *are numbers of nodes of fixed and shifted matching leaf nodes in* t_1 *and* t_2 *respectively* ($A \cap B = \emptyset$). □

The similarity score will be between 0 and 1. Given a set of subtrees from two versions of an XML tree, T_1 and T_2, $\Re(t_{1i}, t_{2j})$ is the similarity score of a pair of *possible matching subtrees* t_{1i} and t_{2j}, where $t_{1i} \in T_1$ and $t_{2j} \in T_2$. Based on the similarity score, we classify the matching subtrees into three types:

- **Identical subtrees** ($\Re(t_{1i},t_{2j})= 1$). For example, subtree t_{20} in T_1 and subtree t_{119} in T_2 are identical subtrees. In the top-down approach, if subtrees X and Y are determined as *identical subtrees*, then we do not need to com-

pare subtrees x_i and y_j, where $x_i \subset X$ and $y_j \subset Y$, as they are also identical subtrees.

- **Unmatching subtrees** ($\Re(t_{1i},t_{2j})= 0$). We say two subtrees are unmatching if they are totally different. For example, subtree t_3 in T_1 and subtree t_{103} in T_2 are unmatching subtrees ($\Re(t_3,t_{103})= 0$). In the top-down approach, if subtrees X and Y are determined as *unmatching subtrees*, then we do not need to compare subtrees x_i and y_j, where $x_i \subset X$ and $y_j \subset Y$, as they are also unmatching subtrees.
- **Matching subtrees** ($0 < \Re(t_{1i},t_{2j})< 1$). For instance, subtree t_{12} in T_1 and subtree t_{112} in T_2 are matching subtrees ($\Re(t_{12},t_{112})= 0.6667$). The higher $\Re(t_{1i}, t_{2j})$ of matching subtree indicates that the subtrees are more similar.

In order to minimize the number of subtree comparisons, we define a *minimum score threshold* θ. If $\Re(t_{1i}, t_{2j}) < \theta$, then we assume that t_{1i} and t_{2j} are *unmatching subtrees*. The value of θ is between 0 and 1. In most cases, the smaller value of θ shall result in better quality of XML deltas. After we are able to determine how similar the possible matching subtrees are, the best matching subtrees can be determined. The formal definition of the best matching subtrees is as follows.

Definition 3.5 [**Best Matching Subtrees**] *Let $t \in T_1$ be a subtree in T_1 and $P \subseteq T_2$ be a set of subtrees in T_2. Also t and $t_i \in P$ are possible matching subtrees $\forall\ 0 < i \leq |P|$. Then t and t_i are the* ***best matching subtrees*** *(denoted by $t \fallingdotseq t_i$) iff $(\Re(t, t_i) > \Re(t, t_j))\ \forall\ 0 < j \leq |P|$ and $i \neq j$.* □

Example 3.4 There are five best matching subtrees in our example: $t_9 \fallingdotseq t_{109}$, $t_{12} \fallingdotseq t_{119}$, $t_{15} \fallingdotseq t_{103}$, $t_{18} \fallingdotseq t_{106}$, and $t_1 \fallingdotseq t_{101}$.

Note that if t_1 and t_2 are not best matching subtrees, then they are denoted by $t_1 \not\fallingdotseq t_2$.

3.3 The Top-down Approach

In this section, we shall present the algorithm for finding best matching subtrees in our top-down approach by using the concepts presented in previous sections. Suppose we have two versions of an XML document shredded in a relational database, T_1 and T_2, and minimum score threshold θ. The first step of finding best matching subtrees in the top-down approach is to compare the root nodes of T_1 and T_2. If T_1 has different node name from T_2, then we assume that T_1 and T_2 are different trees. Consequently, the delta shall consist of a deletion of T_1 and an insertion of T_2. Otherwise, $\Re(T_1,T_2)$ is calculated. Based on Definition 3.4, the similarity score of two subtrees is calculated by using the number of fixed and shifted matching leaf nodes, and the total number of leaf nodes in the both subtrees. The number of fixed and shifted matching leaf nodes can be calculated by using the SQL queries depicted in Figures 5(a) and (b) respectively. Figure 5(c) depicts the SQL query to retrieve the total

```
1  SELECT COUNT(P1.LEAFORDER)
2  FROM LEAFVALUE AS P1 , LEAFVALUE AS P2
3  WHERE
4    P1.DOC_ID = did1 AND P2.DOC_ID = did2 AND
5    P1.PATH_ID = P2.PATH_ID AND
6    P1.LOCALORDER = P2.LOCALORDER AND
7    P1.LEAFVALUE = P2.LEAFVALUE AND
8    P1.SIBLINGORDER BETWEEN
                  minso1 AND maxso1 AND
9    P2.SIBLINGORDER BETWEEN
                  minso2 AND maxso2 AND
10   P1.DEWEY LIKE 'dewey1.%' AND
11   P2.DEWEY LIKE 'dewey2.%' AND
12   SUBSTR(P1.DEWEY, LENGTH(dewey1)) =
         SUBSTR(P2.DEWEY, LENGTH(dewey2))
```

(a) Calculate Number of Fixed Leaf Nodes

```
1 SELECT COUNT(LEAFORDER) AS VALUE
2 FROM LEAFVALUE
3 WHERE DOC_ID = did AND
4   SIBLINGORDER BETWEEN minso AND maxso
```

(c) Calculate Number of Leaf Nodes

```
1 SELECT NODENAME, NODELEVEL,
2        MINSIBORDER, MAXSIBORDER
3 FROM ANCESTORINFO
4 WHERE DOC_ID = did AND
5   MINSIBORDER >= minso AND
6   MAXSIBORDER <= maxso AND
7   NODELEVEL = level+1
```

(d) Get Internal Child Nodes

```
1  SELECT COUNT(P1.LEAFORDER)
2  FROM LEAFVALUE AS P1 , LEAFVALUE AS P2
3  WHERE
4    P1.DOC_ID = did1 AND P2.DOC_ID = did2 AND
5    P1.PATH_ID = P2.PATH_ID AND
6    P1.LEAFVALUE = P2.LEAFVALUE AND
7    P1.SIBLINGORDER BETWEEN minso1 AND maxso1 AND
8    P2.SIBLINGORDER BETWEEN minso2 AND maxso2 AND
9    P1.DEWEY LIKE 'dewey1.%' AND
10   P2.DEWEY LIKE 'dewey2.%' AND
11   SUBSTR(P1.DEWEY, LENGTH(dewey1)) !=
         SUBSTR(P2.DEWEY, LENGTH(dewey2)) AND
12   (P1.LEAFORDER, P2.LEAFORDER) NOT IN (
13      SELECT P1.LEAFORDER, P2.LEAFORDER
14      FROM PATHVALUE AS P1 , PATHVALUE AS P2
15      WHERE
16      P1.DOC_ID = did1 AND P2.DOC_ID = did2 AND
17      P1.PATH_ID = P2.PATH_ID AND
18      P1.LOCALORDER = P2.LOCALORDER AND
19      P1.LEAFVALUE = P2.LEAFVALUE AND
20      P1.SIBLINGORDER BETWEEN
                        minso1 AND maxso1 AND
21      P2.SIBLINGORDER BETWEEN
                        minso2 AND maxso2 AND
22      P1.DEWEY LIKE 'dewey1.%' AND
23      P2.DEWEY LIKE 'dewey2.%' AND
24      SUBSTR(P1.DEWEY, LENGTH(dewey1)) =
            SUBSTR(P2.DEWEY, LENGTH(dewey2)) )
```

(b) Calculate Number of Shifted Leaf Nodes

Fig. 5. Top-down Approach: SQL Queries (1).

number of leaf nodes in a subtree. If $\Re(T_1,T_2) < \theta$, then we also assume that T_1 and T_2 are different trees. If $\Re(T_1,T_2) = 1$, then T_1 and T_2 are identical trees. Consequently, we store the matching information of the root nodes of T_1 and T_2 into database and do not need to do the top-down matching. Otherwise, we store the matching information of the root nodes of T_1 and T_2 into database and start to find best matching subtrees in the top-down fashion. This root node matching is done by using the *rootMatching* algorithm as depicted in Figure 4(a).

The top-down matching is done by using the *TD-findBestMatchingSubtrees* algorithm as depicted in Figure 6. The algorithm works as follows. Suppose we have two root nodes of two subtrees, that is, r_1 and r_2, where $r_1 \in I(T_1)$, $r_2 \in I(T_2)$, $name(r_1)$=$name(r_2)$, and $level(r_1)$=$level(r_2)$. First, we retrieve the child internal nodes of r_1 and r_2. Figure 5(d) depicts the SQL query for retrieving the child internal nodes of an internal node. If at least one of r_1 and r_2 does not have child internal nodes, then the algorithm returns no result. Otherwise, the algorithm starts comparing the child internal nodes of r_1 with the ones of r_2 by calculating the similarity scores. The algorithm tries to compare a child node c_1x of node r_1 with c_2y of r_2 in order to find the most similar one. For each comparison, the algorithm calculates the similarity score of t_1x and t_2y, where c_1x and c_2y are the root nodes of subtrees t_1x and t_2y respectively. After calculating the similarity score, the algorithm checks whether they are identical subtrees ($\Re(t_1x,t_2y)$= 1), unmatching subtrees ($\Re(t_1x,t_2y)< \theta$), or matching subtrees ($\theta \leq \Re(t_1x,t_2y) < 1$).

If $\Re(t_1x, t_2y) < \theta$, then the algorithm assumes that t_1x and t_2y are unmatching subtrees. Consequently, the algorithm will not process the child nodes of

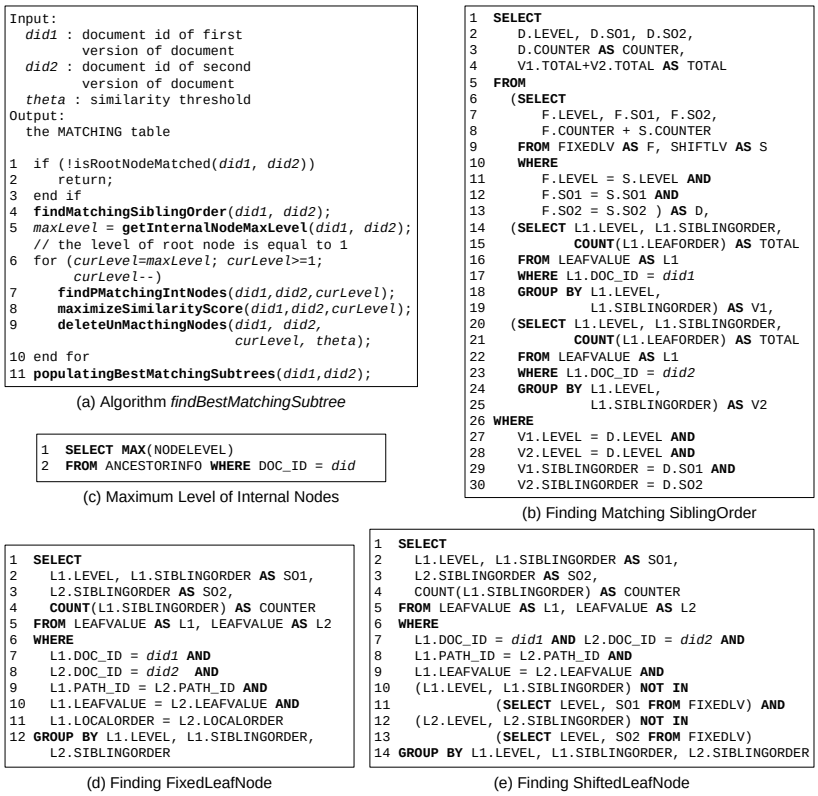
```
Input:
  document id of the first version did1,
  document id of the second version did2,
  parent node in the first version p1,
  parent node in the second version p2,
  threshold theta
Output:
  the MATCHING table

1   list1 = getInternalChildNode(did1, p1);
2   list2 = getInternalChildNode(did2, p2);
3   if (list1 is empty or list2 is empty)
4      return;
5   end if
    // Calculate the similarity score
    // for each pair of subtrees
6   isIdentical = false;
7   Pos = -1;
8   while (list1 is NOT empty and list2 is NOT empty)
9      if (list1[0].name == list2[j].name)
10        score = calculateScore(list1[0], list2[j]);
11        if (score==1.0) // Identical Subtree
12           isIdentical = true;
13           maxScore = score;
14           Pos = j;
15        else if ((score <maxScore) and (score>= theta))
16           maxScore = score;
17           Pos = j;
18        end if
19     end if
20     j++;
21     if (isIdentical or
           (j > sizeOf(list2) and (Pos != -1)))
22        newP1 = list1[0];
23        newP2 = list2[Pos];
24        delete(list1[0]);
25        delete(list2[Pos]);
26        Store (newP1, newP2) in the MATCHING table;
27        if (not isIdentical)
28           findBestMatchingSubtrees(did1, did2,
                                    theta, newP1, newP2);
29        end if
30     else if (j > sizeOf(list2))
31        delete(list1[0]);
32     end if
33  end while
```

Fig. 6. Top-down Approach: Algorithm *TD-findBestMatchingSubtrees*.

c_1x and c_2y. If $\Re(t_1x, t_2y) = 1$, then t_1x and t_2y are identical subtrees. Consequently, the algorithm will store the matching information of these identical subtrees into database. Note that the algorithm will also not process the child nodes of c_1x and c_2y. If $\theta \leq \Re(t_1x,t_2y) < 1$, then t_1x and t_2y are best matching subtrees. Consequently, the algorithm will store the matching information of these best matching subtrees into database and will process the child nodes of c_1x and c_2y in order to find other best matching subtrees in the next level. To process the child nodes of c_1x and c_2y, the algorithm recursively invokes the *TD-findBestMatchingSubtrees* algorithm (line 30, Figure 6).

Finally, the algorithm extends the identical subtrees (if any) by using the SQL query in Figure 4(b). This means that the algorithm maps all identical subtrees in given identical subtrees. Given two root nodes of identical subtrees, the SQL query basically retrieves the internal nodes that are the descendent of these root nodes from the `AncestorInfo` table (lines 19-22, Figure 4(b)). Note that the subtrees in the identical trees must also be identical. Hence, these internal nodes are matched by using their node name (line 16, Figure 4(b)), node level (line 17, Figure 4(b)), and local order (line 18, Figure 4(b)). The information on best matching subtrees are stored in the `Matching` table as depicted in Figure 8(e). The semantics of attributes of the `Matching` table are depicted in Figures 8(a) and (b).

The *TD-findBestMatchingSubtrees* algorithm is a greedy approximation. Once the algorithm determines that subtree rooted at node p in the old version document is matched to subtree rooted at node q in the new version, it will not compare the subtree rooted at node p to subtrees in the new version document nor the subtree rooted at node q to subtrees in the old version document. This greedy approximation may lead the algorithm to result non-optimal delta in some cases. If we do not use the greedy approximation, then the algorithm does $|N| \times |M|$ subtrees comparisons (similarity score calculations) for each level, where $|N|$ and $|M|$ are the numbers of internal nodes that are the child nodes of matching subtrees in the old and new versions of an XML tree respectively. This leads to significant reduction of the performance of the algorithm. Hence, we trade off the result quality for better performance.



```
Input:
  did1 : document id of first
         version of document
  did2 : document id of second
         version of document
  theta : similarity threshold
Output:
  the MATCHING table

1  if (!isRootNodeMatched(did1, did2))
2     return;
3  end if
4  findMatchingSiblingOrder(did1, did2);
5  maxLevel = getInternalNodeMaxLevel(did1, did2);
   // the level of root node is equal to 1
6  for (curLevel=maxLevel; curLevel>=1;
        curLevel--)
7     findPMatchingIntNodes(did1,did2,curLevel);
8     maximizeSimilarityScore(did1,did2,curLevel);
9     deleteUnMacthingNodes(did1, did2,
                            curLevel, theta);
10 end for
11 populatingBestMatchingSubtrees(did1,did2);
```

(a) Algorithm *findBestMatchingSubtree*

```
1  SELECT MAX(NODELEVEL)
2  FROM ANCESTORINFO WHERE DOC_ID = did
```

(c) Maximum Level of Internal Nodes

```
1  SELECT
2     D.LEVEL, D.SO1, D.SO2,
3     D.COUNTER AS COUNTER,
4     V1.TOTAL+V2.TOTAL AS TOTAL
5  FROM
6    (SELECT
7        F.LEVEL, F.SO1, F.SO2,
8        F.COUNTER + S.COUNTER
9     FROM FIXEDLV AS F, SHIFTLV AS S
10    WHERE
11       F.LEVEL = S.LEVEL AND
12       F.SO1 = S.SO1 AND
13       F.SO2 = S.SO2 ) AS D,
14   (SELECT L1.LEVEL, L1.SIBLINGORDER,
15           COUNT(L1.LEAFORDER) AS TOTAL
16    FROM LEAFVALUE AS L1
17    WHERE L1.DOC_ID = did1
18    GROUP BY L1.LEVEL,
19             L1.SIBLINGORDER) AS V1,
20   (SELECT L1.LEVEL, L1.SIBLINGORDER,
21           COUNT(L1.LEAFORDER) AS TOTAL
22    FROM LEAFVALUE AS L1
23    WHERE L1.DOC_ID = did2
24    GROUP BY L1.LEVEL,
25             L1.SIBLINGORDER) AS V2
26 WHERE
27    V1.LEVEL = D.LEVEL AND
28    V2.LEVEL = D.LEVEL AND
29    V1.SIBLINGORDER = D.SO1 AND
30    V2.SIBLINGORDER = D.SO2
```

(b) Finding Matching SiblingOrder

```
1  SELECT
2    L1.LEVEL, L1.SIBLINGORDER AS SO1,
3    L2.SIBLINGORDER AS SO2,
4    COUNT(L1.SIBLINGORDER) AS COUNTER
5  FROM LEAFVALUE AS L1, LEAFVALUE AS L2
6  WHERE
7    L1.DOC_ID = did1 AND
8    L2.DOC_ID = did2  AND
9    L1.PATH_ID = L2.PATH_ID AND
10   L1.LEAFVALUE = L2.LEAFVALUE AND
11   L1.LOCALORDER = L2.LOCALORDER
12 GROUP BY L1.LEVEL, L1.SIBLINGORDER,
     L2.SIBLINGORDER
```

(d) Finding FixedLeafNode

```
1  SELECT
2    L1.LEVEL, L1.SIBLINGORDER AS SO1,
3    L2.SIBLINGORDER AS SO2,
4    COUNT(L1.SIBLINGORDER) AS COUNTER
5  FROM LEAFVALUE AS L1, LEAFVALUE AS L2
6  WHERE
7    L1.DOC_ID = did1 AND L2.DOC_ID = did2 AND
8    L1.PATH_ID = L2.PATH_ID AND
9    L1.LEAFVALUE = L2.LEAFVALUE AND
10   (L1.LEVEL, L1.SIBLINGORDER) NOT IN
11            (SELECT LEVEL, SO1 FROM FIXEDLV) AND
12   (L2.LEVEL, L2.SIBLINGORDER) NOT IN
13            (SELECT LEVEL, SO2 FROM FIXEDLV)
14 GROUP BY L1.LEVEL, L1.SIBLINGORDER, L2.SIBLINGORDER
```

(e) Finding ShiftedLeafNode

Fig. 7. Bottom Up Approach: Algorithm *BU-findBestMatchingSubtree* and SQL Queries.

Observe that the top-down approach has two drawbacks. First, the detected delta may not be optimal delta in some cases as it uses a greedy approximation. Second, the first phase ("finding best matching subtrees") in the top-down approach is a time consuming process. In the next section, we shall present another approach that is able to overcome these drawbacks.

3.4 The Bottom-up Approach

In this section, we elaborate the first phase of the bottom-up approach to find best matching subtrees in T_1 and T_2 by using the concepts presented in the former section. The algorithm for determining the best matching subtrees in T_1 and T_2 is shown in Figure 7(a). The *BU-findBestMatchingSubtree* algorithm is a bottom-up algorithm as it starts finding best matching subtrees from lower levels to the root node. Here we use an example to illustrate the algorithm.

To find best matching subtrees in T_1 and T_2, first we check whether the root nodes of T_1 and T_2 have the same name. If they have different node name, then we assume that T_1 and T_2 are different. Consequently, the delta will consist of a deletion of T_1 and an insertion of T_1. Otherwise, the algorithm shall find the best matching sibling orders in T_1 and T_2. The SQL query for retrieving matching sibling orders in T_1 and T_2 is depicted in Figure 7(b). The `FIXEDLV` and `SHIFTLV` relations are two sets of *fixed* and *shifted* matching leaf nodes

TempSO (Level, SO1, SO2, Counter, Total)
TempMatching (MinSO1, MaxSO1, MinSO2, MaxSO2, Level, LO1, LO2, Flag, Counter, Total, Score)
Matching (DID1, DID2, MinSO1, MaxSO1, MinSO2, MaxSO2, Level, LO1, LO2, Score)

(a) Attributes of The TempSO and Matching Tables

Attribute	Description
DID1	The document id of the first document
DID2	The document id of the second document
LEVEL	The node's level
MINSO1	The minimum sibling order of an internal node in the first version
MAXSO1	The maximum sibling order of an internal node in the first version
MINSO1	The minimum sibling order of an internal node in the second version
MAXSO1	The maximum sibling order of an internal node in the second version
LO1	The local order of a node in the first version
LO2	The local order of a node in the second version
SO1	The sibling order of a leaf node in the first version
SO2	The sibling order of a leaf node in the second version
Flag	Annotation indicating that a subtree is moved to different parent node
Counter	The number of matching leaf nodes
Total	The total number of leaf nodes
Score	The similarity score

(b) Description of Attributes

Level	SO1	SO2	Counter	Total
2	1	1	2	2
3	2	4	2	4
3	4	4	4	4
3	6	2	2	4
3	6	4	2	4
3	8	6	2	4
3	8	8	4	4
4	5	5	1	3
4	7	3	2	3
4	9	9	4	4

(c) TempSO Table

Level	MinSO1	MaxSO1	MinSO2	MaxSO2	LO1	LO2	Flag	Counter	Total	Score
3	5	5	5	5	3	3	0	1	3	0.3333
3	7	7	3	3	3	3	0	2	3	0.6666
3	9	9	9	9	3	3	0	4	4	1.0000
2	2	3	4	5	2	3	0	2	7	0.2857
2	4	5	4	5	3	3	0	5	7	0.7142
2	6	7	2	3	4	2	0	4	7	0.5714
2	6	7	4	5	4	3	0	2	6	0.3333
2	8	9	6	7	5	4	0	2	7	0.2857
2	8	9	8	9	5	5	0	8	8	1.0000
1	1	9	1	9	1	1	0	19	31	0.6129

(d) TempMatching Table

DID1	DID2	Level	Min SO1	Max SO1	Min SO2	Max SO2	LO1	LO2	Score
1	2	3	5	5	5	5	3	3	0.3333
1	2	3	7	7	3	3	3	3	0.6666
1	2	3	9	9	9	9	3	3	1.0000
1	2	2	4	5	4	5	3	3	0.7142
1	2	2	6	7	2	3	4	2	0.5714
1	2	2	8	9	8	9	5	5	1.0000
1	2	1	1	9	1	9	1	1	0.6129

(e) Matching Table

Fig. 8. Bottom Up Approach: The `TempSO`, `TempMatching`, and `Matching` Tables.

respectively. The fixed matching leaf nodes and shifted matching leaf nodes can be determined by using the SQL query depicted in Figures 7(d) and (e) respectively. The matching siblings orders will be stored in the `TempSO` table as depicted in Figure 8(c). The semantics of the attributes of the `TempSO` table are depicted in Figures 8(a) and (b).

Next, we determine the deepest level $maxLevel$ of internal nodes in T_1 and T_2 by using the SQL query as depicted in Figure 7(c). For each level $curLevel$ starting from level $maxLevel$ to the level of the root nodes, the algorithm starts finding the best matching subtrees. First, the algorithm shall find the possible matching internal nodes at which the possible matching subtrees are rooted. The SQL query in Figure 9(a) is used to retrieve the root nodes of the possible matching subtrees at level $curLevel$ based on Definition 3.3. We use the information on matching sibling orders to match the parent nodes of leaf nodes in matching sibling orders. That is, the subtrees rooted at these parent nodes will have at least one matching leaf node. We store the results into the `TempMatching` table as depicted in Figure 8(d). The semantics of the attributes of the `TempMatching` table are depicted in Figures 8(a) and (b). The `Flag` attribute of the `TempMatching` table is initially set to "0". The usage of the `Flag` attribute shall be discussed later.

The next step is to maximize the similarity scores of the possible matching internal nodes at level $curLevel$. This is because we may have some subtrees and sibling orders at $(curLevel + 1)$ in T_1 that can be matched to more than one subtree and sibling order in T_2 respectively, and vice versa. For example, there are several possible matching subtrees in level 2: $S_3 \simeq S_{109}$, $S_9 \simeq S_{109}$, $S_{15} \simeq S_{109}$, $S_{15} \simeq S_{103}$, $S_{20} \simeq S_{114}$, and $S_{20} \simeq S_{119}$. There is a matching sibling order in level 2: $s_{11} \Leftrightarrow s_{21}$. The algorithm needs to find what matching combination of these possible matching subtrees and matching sibling orders such that $\Re(1, 101)$ is maximized. There are six possible matching combinations of these possible matching subtrees and matching sibling orders as

```
1  SELECT
2     A1.NODELEVEL,
3     A1.MINSIBORDER AS MINSO1,
4     A1.MAXSIBORDER AS MAXSO1,
5     A2.MINSIBORDER AS MINSO2,
6     A2.MAXSIBORDER AS MAXSO2,
7     SUM(T.COUNTER) AS COUNTER,
8     SUM(T.TOTAL) AS TOTAL
9  FROM
10    ANCESTORINFO AS A1, ANCESTORINFO AS A2,
11    TEMPSO AS T
12 WHERE
13    A1.DOC_ID = did1 AND
14    A2.DOC_ID = did2 AND
15    T.SO1 BETWEEN A1.MINSIBORDER AND
            A1.MAXSIBORDER AND
16    T.SO2 BETWEEN A2.MINSIBORDER AND
            A2.MAXSIBORDER AND
17    A1.NODELEVEL = A2.NODELEVEL AND
18    A1.NODENAME = A2.NODENAME AND
19    A1.NODELEVEL = level
20 GROUP BY A1.NODELEVEL,
     A1.MINSIBORDER, A1.MAXSIBORDER,
     A2.MINSIBORDER, A2.MAXSIBORDER
```

(a) Finding Possible Matching Internal Node

```
Input :
   document id of first version of document did1,
   document id of second version of document did2,
   level curLevel,
   the MATCHING and TEMPSO tables
Output: the MATCHING table

   // find non one-to-one matching relation
   // groupped by the parent nodes
1  Data = getNon121Matching(did1, did2, curLevel);
2  FOR EACH group d IN Data
     // generate the scoreMatrix
3    scoreMatrix = generateScoreMatrix(d);
     // Start finding the maximum score
4    (maxScore, maxScoreCombination) =
               bestCombinationFinder(scoreMatrix);
     // delete the corresponding tuples in
     // the MATCHING table that are
     // not used in getting maximum score
5    annotateUnusedTuple(maxScoreCombination);
6    deleteUnusedTuple(maxScoreCombination);
7  END FOR
```

(b) The *maximizeSimilarityScore* Algorithm

```
Input : scoreMatrix
Output: maximumScore and maxScoreCombination

1  SET row TO the numbers of row of scoreMatrix;
2  SET column TO the numbers of column of scoreMatrix;
3  INITIALIZE backTrackMatrix[row][column];
4  SET maxScore TO zero;
5  SET maxScoreCombination TO empty;
6  FOR i = 1 TO row DO
7    FOR j = 1 TO column DO
8      IF  (scoreMatrix[i][j].matchScore > 0) THEN
9        IF  ( i = 1 ) THEN  // if it it the first row
10         ADD j TO backTrackMatrix[i][j];
11         SET scoreMatrix[i][j].maxScore TO
                                scoreMatrix[i][j].matchScore;
12         IF (scoreMatrix[i][j].matchScore > maxScore) THEN
13           maxScore = scoreMatrix[i][j].matchScore;
14           maxScoreCombination = backTrackMatrix[i][j];
15         END IF
16       ELSE
           // Use score in row i-1 to calculate score in row i
17         FOR k = 1 TO column DO
18           IF ((j is not in backTrackMatrix[i-1][k]) AND
                (scoreMatrix[i][j].maxScore <
                     (scoreMatrix[i][j].matchScore +
                      scoreMatrix[i-1][x].maxScore))) THEN
19               scoreMatrix[i][j].maxScore =
                        scoreMatrix[i][j].matchScore +
                        scoreMatrix[i-1][x].maxScore;
20               SET backTrackMatrix[i][j] TO
                                 backTrackMatrix[i-1][x];
21               ADD j TO backTrackMatrix[i][j];
22             END IF
23             IF ((j != k) AND
                  (scoreMatrix[i][j].maxScore <
                        (scoreMatrix[i][j].matchScore +
                         scoreMatrix[i-1][x].matchScore))) THEN
24                scoreMatrix[i][j].maxScore =
                         scoreMatrix[i][j].matchScore +
                         scoreMatrix[i-1][x].matchScore;
25                RESET backTrackMatrix[i][j];
26                ADD j TO backTrackMatrix[i][j];
27              END IF
28           END FOR
29           IF (scoreMatrix[i][j].matchScore > maxScore) THEN
30             maxScore = scoreMatrix[i][j].matchScore;
31             maxScoreCombination = backTrackMatrix[i][j];
32           END IF
33       END IF
34     END IF
35   END FOR
36 END FOR
37 RETURN maxScore and maxScoreCombination;
```

(c) The bestCombinationFinder Algorithm

Fig. 9. SQL Query, and The *maximizeSimilarityScore* and *bestCombinationFinder* Algorithms.

follows. First, $s_{11} \Leftrightarrow s_{21}$, $S_3 \backsimeq S_{109}$, $S_{15} \backsimeq S_{103}$, and $S_{20} \backsimeq S_{114}$. Second, $s_{11} \Leftrightarrow s_{21}$, $S_3 \backsimeq S_{109}$, $S_{15} \backsimeq S_{103}$, and $S_{20} \backsimeq S_{119}$. Third, $s_{11} \Leftrightarrow s_{21}$, $S_9 \backsimeq S_{109}$, $S_{15} \backsimeq S_{103}$, and $S_{20} \backsimeq S_{114}$. Fourth, $s_{11} \Leftrightarrow s_{21}$, $S_9 \backsimeq S_{109}$, $S_{15} \backsimeq S_{103}$, and $S_{20} \backsimeq S_{119}$. Fifth, $s_{11} \Leftrightarrow s_{21}$, $S_{15} \backsimeq S_{109}$, and $S_{20} \backsimeq S_{114}$. Sixth, $s_{11} \Leftrightarrow s_{21}$, $S_{15} \backsimeq S_{109}$, and $S_{20} \backsimeq S_{119}$. The matching combination that results the maximum $\Re(1, 101)$ is $s_{11} \Leftrightarrow s_{21}$, $S_9 \backsimeq S_{109}$, $S_{15} \backsimeq S_{103}$, and $S_{20} \backsimeq S_{119}$. We use dynamic programming to determine the *best matching configuration* that maximizes the similarity score of the possible matching internal nodes at level *curLevel*. Dynamic programming is chosen as we need to find *best matching configuration* from several matching configurations. The *maximizeSimilarityScore* algorithm is depicted in Figure 9(b). The first step of the algorithm is to find non one-to-one matching relations at level *curLevel* and group them according to the parent nodes. The *maximizeSimilarityScore* algorithm shall invoke the *bestCombinationFinder* algorithm as depicted in Figure 9(c) to find the *best matching configuration*. Note that the *bestCombinationFinder* algorithm is motivated by the Smith-Waterman algorithm [14] for sequence alignments. The *bestCombinationFinder* returns the maximum score that can be achieved and the best matching configuration that maximizes the similarity score. Next, the *maximizeSimilarityScore* algorithm uses the best matching configuration returned by the *bestCombinationFinder* algorithm to annotate

Notation	Description
N	A set of inserted internal nodes n_i, where $N \subseteq I(T_2)$
Y	A set of inserted leaf nodes y_i, where $Y \subseteq L(T_2)$
D	A set of deleted internal nodes d_i, where $D \subseteq I(T_1)$
Z	A set of deleted leaf nodes z_i, where $Z \subseteq L(T_1)$
U	A set of updated leaf nodes u_i
U_a	A set of absolute updated leaf nodes u_{ai}, where $U_a \subseteq U$
U_r	A set of relative updated leaf nodes u_{ri}, where $U_r \subseteq U$

Fig. 10. Notations.

and delete the corresponding tuples of nodes that are not used in the best matching configuration (lines 5-6, Figure 9(b)). The algorithm annotates the root nodes of the possible matching subtrees at level $(curLevel + 1)$ whose parents are not used in the best matching configuration by setting the `Flag` attribute in the `TempMatching` table to "1". The annotations mean that these subtrees may be moved to different parent nodes.

Then the *BU-findBestMatchingSubtree* algorithm deletes the root nodes of subtrees at level *curLevel* that are categorized as unmatching subtrees. After the algorithm determines the best matching subtrees up to the root nodes of T_1 and T_2, it populates the best matching subtrees from the `TempMatching` table. The best matching subtrees in T_1 and T_2 are stored in the `Matching` table. Note that the `TempMatching` table also stores the corresponding tuples of the root nodes of the subtrees that are suspected as moved subtrees. The semantics of the attributes of the `Matching` table are Figures 8(a) and (b). The `Matching` table storing the best matching subtrees of our example is depicted in Figure 8(e).

The *bottom-up approach* is able to find the best matching subtrees in T_1 and T_2. Intuitively, a set S is maximized if all subsets s_i of S are maximized. That is, if subtrees at level $l+1$ have maximum similarity score, then subtrees at level l will also have maximum similarity score. Therefore, the *bottom-up approach* is able to find the best matching subtrees by maximizing the similarity scores of the subtrees from the lower levels to the root nodes.

4 Detecting the Changes

After we are able to identify best matching subtrees in T_1 and T_2, we are ready to detect the changes between T_1 and T_2 by using the information on the best matching subtrees. There are seven types of changes considered in this article: *insertion of internal nodes*, *insertion of leaf nodes*, *deletion of internal nodes*, *deletion of leaf nodes*, *content update of leaf nodes*, *move among siblings*, and *move to different parent nodes*. In this section, we shall discuss the concepts that we shall use in finding the changes. We also present the SQL queries based on the properties to find the XDeltas.

We now define the notion of *parent node* with respect to relational schema that we use to store XML documents as follows.

Definition 4.1 [**Parent Nodes**] *Let* $minsiborder(x)$ *and* $maxsiborder(x)$ *be the minimum sibling order and maximum sibling order of an internal node* x *respectively. Node* i *is a* ***parent node*** *of a node* n *(denoted by* $parent(n)$*) iff* $level(n) = level(i) + 1$*, and satisfies:*

- *if* n *is a* ***leaf node****, then* $minsiborder(i) \leq siborder(n) \leq maxsiborder(i)$.
- *if* n *is an* ***internal node****, then* $minsiborder(i) \leq minsiborder(n)$ *and* $maxsiborder(i) \geq maxsiborder(n)$. □

Figure 10 depicts the notations that will be used in our discussion.

4.1 Types of Changes

In this section, we shall elaborate the properties for detecting types of changes in turns.

4.1.1 Insertion

There are two types of insertions: *insertion of internal nodes* and *insertion of leaf nodes*.

Insertion of Internal Nodes

Intuitively, the inserted internal nodes are internal nodes that are in the new version (T_2), but not in the old version (T_1). Hence, the inserted internal nodes must not be the root nodes of best matching subtrees as these nodes are in both versions. Formally,

Definition 4.2 [**Inserted Internal Nodes**] *Node* n *is an* ***inserted internal node*** *if* $n \in I(T_2)$*, and* $\forall j_x \in I(T_1)$ *such that* $j_x \neq n$. □

Example 4.1 We have two best matching subtrees at level 2 ($t_9 \fallingdotseq t_{109}$ and $t_{15} \fallingdotseq t_{103}$). The node 114 that is the root node of subtree t_{114} in T_2 is an inserted internal node as the conditions in Definition 4.2 are satisfied.

An internal node i in tree T is identified by four properties of node i (node level, minimum sibling order, and maximum sibling order). By using Definition 4.2 and these properties of internal nodes, we are able to find inserted internal nodes.

Insertion of Leaf Nodes

The *new* leaf nodes are only available in the new version of an XML tree (T_2). We observed that there are two types of inserted leaf nodes as follows.

- **Inserted leaf nodes in the newly inserted subtrees**. These inserted leaf nodes must be the child nodes of inserted internal nodes. Note that the inserted internal nodes are the root nodes of inserted subtrees. Consider

two versions of an XML document depicted in Figure 1. The leaf nodes with identifier 115, 116, and 118 belong to the newly *inserted subtree* rooted at node 114.

- **Inserted leaf nodes in the best matching subtrees**. The parent nodes of these inserted leaf nodes are the root nodes of best matching subtrees. Note that the best matching subtrees are in the old and new versions of XML documents. Consider two versions of an XML document depicted in Figure 1. The leaf node with identifier 108 is also inserted in the new version. The parent node of node 108 is node 106 which are matched to node 18 ($t_{18} \fallingdotseq t_{106}$).

Definition 4.3 [**Inserted Leaf Nodes**] $y \in L(T_2)$ *is an* ***inserted leaf node*** *if the following conditions are satisfied:*

- *if* y *is in a* ***newly inserted subtree****, then* $parent(y) = n_i$*, where* $n_i \in N$*,*
- *if* y *is in a* ***best matching subtree****, then* $parent(y) = i_2$*, and* $\forall a_x \in L(T_1)$ *such that* $(a_x \not\leftrightarrow y)$*, where* $parent(a_x) = i_1$ *and* $t_{i_1} \fallingdotseq t_{i_2}$*.* □

Note that by using Definition 4.3, we also detect the updated leaf nodes as they can be decomposed into pairs of deleted and inserted of leaf nodes. For example, we have $t_{15} \fallingdotseq t_{103}$. Node 105 which should be an updated leaf node is detected as an inserted leaf node.

4.1.2 Deletion

There are also two types of deletions: *deletion of internal nodes* and *deletion of leaf nodes*.

Deletion of Internal Nodes

The deleted internal nodes can be determined by using the same intuitions as one for finding inserted internal nodes. The deleted internal nodes are in the old version (T_1), but not in the new version (T_2). The deletion of an internal node is formally defined as follows.

Definition 4.4 [**Deleted Internal Nodes**] *Node* j *is a* ***deleted internal node*** *if* $j \in I(T_1)$*, and* $\forall i_x \in I(T_2)$ *such that* $i_x \not\fallingdotseq j$*.* □

Example 4.2 We have two best matching subtrees at level 2 ($t_9 \fallingdotseq t_{109}$ and $t_{15} \fallingdotseq t_{103}$). The node 3 that is the root node of subtree t_3 in T_1 is a deleted internal node as the conditions in Definition 4.4 are satisfied.

Deletion of Leaf Nodes

Intuitively, the *deleted* leaf nodes are only available in the old version of an XML tree (T_1). We noticed that there are two types of deleted leaf nodes as follows.

- **Deleted leaf nodes in the deleted subtrees**. The parent nodes of these deleted leaf nodes are deleted internal nodes which are the root nodes of deleted subtrees. For example, the leaf nodes with identifier 4, 5, 7, and 8 belong to the *deleted subtree* rooted at node 3.
- **Deleted leaf nodes in the best matching subtrees**. The parent nodes of these deleted leaf nodes are the root nodes of best matching subtrees. Consider two versions of an XML document depicted in Figure 1. The leaf node with identifier 13 is also deleted. This leaf node is in the best matching subtrees $t_{12} \nsim t_{112}$.

Definition 4.5 [**Deleted Leaf Nodes**] $z \in L(T_1)$ *is an* ***deleted leaf node*** *if the following conditions are satisfied:*

- *if* z *is in a* ***deleted subtree****, then* $parent(z) = d_i$*, where* $d_i \in D$*,*
- *if* z *is in a* ***best matching subtree****, then* $parent(z) = i_1$*, and* $\forall b_x \in L(T_2)$ *such that* $(b_x \not\leftrightarrow z)$*, where* $parent(b_x) = i_2$ *and* $t_{i_1} \nsim t_{i_2}$*.* □

Note that by using Definition 4.5, we also detect the updated leaf nodes as they can be decomposed into pairs of deleted and inserted of leaf nodes. For example, we have $t_{15} \nsim t_{103}$. Node 17 which should be an updated leaf node is detected as a deleted leaf node.

4.1.3 Update

Intuitively, an updated node is available in the first and second versions, but its value is different. As the updated leaf nodes are detected as pairs of deleted and inserted leaf nodes by using Definitions 4.5 and 4.3 respectively, we are able to find the updated leaf nodes from two sets of leaf nodes: *inserted leaf nodes* and *deleted leaf nodes*. In addition, we also need the information on best matching subtrees in order to guarantee that the updated leaf nodes are in best matching subtrees. Note that we only consider the update of the content of leaf nodes. The modification of the name of a node is detected as a pair of deletion and insertion.

As the position among siblings in ordered XML is important, the update operation can be classified into two types: *absolute update operation* and *relative update operation*. In the absolute update operation, only the content value of an updated leaf node is changed, while its position among siblings remains the same. In relative update operation, the content value and position among siblings of an updated leaf node are changed.

Definition 4.6 [**Absolute Updated Leaf Nodes**] *Let* $Z_a \subseteq Z$ *be a set of deleted leaf nodes and* $Y_a \subseteq Y$ *be a set of inserted leaf nodes. Let* $z_a \in Z_a$ *and* $y_a \in Y_a$ *be a deleted leaf node and an inserted leaf node respectively. Let* $localorder(x)$ *be the local order of a leaf node* x*. A leaf node* u_a *is an* ***absolute updated leaf node*** *decomposed as a deletion of leaf node* z_a *and an insertion of leaf node* y_a *if* $name(z_a) = name(y_a)$*,* $level(z_a) = level(y_a)$*,*

$localorder(z_a) = localorder(y_a)$, $value(z_a) \neq value(y_a)$, *and* $(parent(z_a) \nsim parent(y_a))$. □

Example 4.3 The subtrees rooted at node 15 in T_1 and node 103 in T_2 are best matching subtrees. Node 17 is updated from "Assoc Prof" to "Prof". This update operation is classified into absolute update operation as $name(\ell_{17}) = name(\ell_{105})$ ("rank"), $level(\ell_{17}) = level(\ell_{105})$ ("3"), $localorder(\ell_{17}) = localorder(\ell_{105})$ ("2"), $value(\ell_{17}) \neq value(\ell_{105})$ ("Assoc Prof" ≠ "Prof"), and $(t_{15} \nsim t_{103})$.

Definition 4.7 [**Relative Updated Leaf Nodes**] *Let $Z_r \subseteq Z$ be a set of deleted leaf nodes and $Y_r \subseteq Y$ be a set of inserted leaf nodes. Let $z_r \in Z_r$ and $y_r \in Y_r$ be a deleted leaf node and an inserted leaf node respectively. A leaf node u_r is a* ***relative updated leaf node*** *decomposed as a deletion of leaf node z_r and an insertion of leaf node y_r if $name(z_r) = name(y_r)$, $level(z_r) = level(y_r)$, $localorder(z_r) \neq localorder(y_r)$, $value(z_r) \neq value(y_r)$, $(parent(z_r) \nsim parent(y_r))$, $Z_r \cap Z_a = \emptyset$, and $Y_r \cap Y_a = \emptyset$.* □

Example 4.4 The subtrees rooted at node 23 in T_1 and node 122 in T_2 are best matching subtrees. Node 24 is updated from "Indexing" to "XML Indexing". This update operation is classified into relative update operation as $name(\ell_{24}) = name(\ell_{124})$ ("interest"), $level(\ell_{24}) = level(\ell_{124})$ ("4"), $localorder(\ell_{24}) \neq localorder(\ell_{124})$ ("1" ≠ "2"), $value(\ell_{24}) \neq value(\ell_{124})$ ("Indexing" ≠ "XML Indexing"), and $(t_{23} \nsim t_{122})$.

4.1.4 Move

The move operations are classified into two categories. First, the moved node changes its position among its siblings in the XML tree. That is, before and after the move operation, it has *same* parent but different position among its siblings. Second, the node (subtree) is moved to be the child of a different parent. That is, before and after the move operation, it has *different* parents, and may have different position among its siblings. These two move operations are formally defined as follows.

Definition 4.8 [**Moved Internal Node**] *Let S_1 and S_2 be two subtrees rooted at nodes $i_1 \in D$ and $i_2 \in N$ respectively. Subtree S_1 is moved if $(S_1 \nsim S_2)$ and satisfies: (a) if S_1 is* ***moved among its siblings****, then $localorder(i_1) \neq localorder(i_2)$, and $(parent(i_1) \nsim parent(i_2))$, (b) if S_1 is* ***moved to different parent node****, then $(parent(i_1) \not\sim parent(i_2))$.* □

Definition 4.8 defines move operations of internal nodes. The same intuitions as in Definition 4.8 is used to define move operations of leaf nodes.

Definition 4.9 [**Moved Leaf Node**] *Let ℓ_1 and ℓ_2 be two leaf nodes where $\ell_1 \in Z$ and $\ell_2 \in Y$ respectively. ℓ_1 is moved if $(\ell_1 \leftrightarrow \ell_2)$ and satisfies: (a) if ℓ_1 is* ***moved among its siblings****, then $localorder(\ell_1) \neq localorder(\ell_2)$, and*

$(parent(\ell_1) \eqslantless parent(\ell_2))$, *(b) if* ℓ_1 *is* ***moved to different parent node****, then* $(parent(\ell_1) \not\eqslantless parent(\ell_2))$. □

From Definitions 4.8 and 4.9, we notice that a node is moved among its siblings if its local order is changed. The local order of a node may be changed because there are insertions/deletions of its sibling nodes. We observed that a deletion of node a, that has local order equal to k, will decrease the local orders of its siblings, that have local order greater than k, by one. Another observation is that an insertion of node b to be the k-th child of a parent node p will increase the local orders of the child nodes of node p, that have local order greater than or equal to k, by one. Note that we are not interested in the changes on the local orders because of insertions/deletions of its sibling nodes. Hence, we need to determine the nodes that are *really* moved among their siblings. We are able to determine these moved nodes by using the above observations for simulating the insertions/deletions of sibling nodes that affect on the local orders. We shall elaborate further in the subsequent sections.

4.2 SQL Queries

In this section, we shall present the SQL queries that are used to detect the changes. The SQL queries are written based on the definitions presented in the previous section.

4.2.1 Insertion

Insertion of Internal Nodes

The inserted internal nodes can be found by using the `AncestorInfo` and `Matching` tables. As the internal nodes in the `AncestorInfo` table are identified by their node level, minimum sibling orders, and maximum sibling orders, the `Level`, `MinSibOrder`, and `MaxSibOrder` attributes in the `AncestorInfo` table are used. We also use the `Doc_ID` attribute of the `AncestorInfo` table as inserted internal nodes must be in the second version of an XML document. The `DID1`, `DID2`, `Level`, `MinSO2`, and `MaxSO2` attributes of the `Matching` table are used to find inserted internal nodes. The SQL query depicted in Figure 11(a) (denoted by `SQL-01`) detects a set of newly inserted internal nodes. The *did1* and *did2* refer to the document id of the old and new versions of an XML document respectively.

The `SQL-01` is able to find all inserted internal nodes. Suppose we have an internal node n. If n is in the new version, then the condition (line 6) in the `WHERE`-clause of `SQL-01` is true. If n has a corresponding node in the old version, then the information on n will be the query result of sub query of `SQL-01` (lines $8-10$). Consequently, the condition (lines $7-10$) in the `WHERE`-clause of `SQL-01` is false. But if n has no corresponding node in the old version, then the information on n will not be the query result of sub query of `SQL-01`

```
1  SELECT DISTINCT
2    did1, did2, NODENAME, NODELEVEL,
3    MINSIBORDER, MAXSIBORDER, LOCALORDER
4  FROM ANCESTORINFO
5  WHERE
6    DOC_ID = did2 AND
7    (NODELEVEL,MINSIBORDER,MAXSIBORDER) NOT IN
8       (SELECT LEVEL, MINSO2,MAXSO2
9        FROM MATCHING
10       WHERE DID1 = did1 AND DID2 = did2)
```

(a) Detecting Inserted Internal Nodes.

```
1  SELECT DISTINCT
2    did1, did2, PV.SIBLINGORDER, PV.PATH_ID,
3    PV.LEAFVALUE, PV.LOCALORDER,
4    PV.LEVEL
5  FROM LEAFVALUE AS PV, INS_INT AS II
6  WHERE
7    PV.DOC_ID = did1 AND
8    II.DID1 = did1 AND II.DID2 = did2 AND
9    PV.SIBLINGORDER BETWEEN
           II.MINSO AND II.MAXSO AND
10   PV.LEVEL = II.LEVEL+1
```

(b) Detecting Inserted Leaf Nodes (1).

```
1  SELECT
2     did1, did2, PV.LEVEL, PV.SIBLINGORDER,
3     PV.PATH_ID, PV.LEAFVALUE, PV.LOCALORDER
4  FROM LEAFVALUE AS PV,
5    (SELECT DISTINCT
6        M.MINSO1, M.MAXSO1, M.MINSO2, M.MAXSO2,
7        PV.PATH_ID, PV.LEAFVALUE
8     FROM MATCHING AS M, LEAFVALUE AS PV
9     WHERE
10       M.DID1 = did1 AND M.DID2 = did2 AND
11       PV.DOC_ID = did2 AND
12       PV.LEVEL = M.LEVEL+1 AND
13       PV.SIBLINGORDER BETWEEN M.MINSO2 AND M.MAXSO2
14    EXCEPT ALL
15    SELECT DISTINCT
16       M.MINSO1, M.MAXSO1, M.MINSO2, M.MAXSO2,
17       PV.PATH_ID, PV.LEAFVALUE
18    FROM MATCHING AS M, LEAFVALUE AS PV
19    WHERE
20       M.DID1 = did1 AND M.DID2 = did2 AND
21       PV.DOC_ID = did1 AND
22       PV.LEVEL = M.LEVEL+1 AND
23       PV.SIBLINGORDER BETWEEN M.MINSO1 AND M.MAXSO1) AS D
24 WHERE
25    PV.DOC_ID = did2 AND
26    PV.SIBLINGORDER BETWEEN D.MINSO2 AND D.MAXSO2 AND
27    PV.PATH_ID = D.PATH_ID AND
28    PV.LEAFVALUE = D.LEAFVALUE
```

(c) Detecting Inserted Leaf Nodes (2).

Fig. 11. Detecting Changes: SQL Queries (1).

(lines $8-10$). Consequently, the condition (lines $7-10$) in the `WHERE`-clause of `SQL-01` is true. Therefore, node n that is in the result of `SQL-01` is an inserted node.

The SQL query depicted in Figure 11(a) is able to return all inserted internal nodes. The result of this SQL query is stored in the `INS_INT` table as depicted in Figure 13(a). The semantics of attributes of the `INS_INT` table are depicted in Figures 12(d) and (e).

Insertion of Leaf Nodes

Recall that there are two types of inserted leaf nodes: *inserted leaf nodes in newly inserted subtrees*, and *inserted leaf nodes in best matching subtrees*. We use the `Doc_ID`, `Path_ID`, `Level`, and `SiblingOrder` attributes of the `LeafValue` table to detect the inserted leaf nodes in inserted subtrees. The `DID1`, `DID2`, `Level`, `MinSO` and `MaxSO` attributes of the `INS_INT` table are also used. To detect the inserted leaf nodes in best matching subtrees we use the same attributes of the `LeafValue` table as the ones for detecting the inserted leaf nodes in inserted subtrees. We need to use the `DID1`, `DID2`, `Level`, `MinSO1`, `MaxSO1`, `MinSO2`, and `MaxSO2`. Note that by using these properties, we also detect the updated leaf nodes as they can be decomposed into a pair of deleted and inserted leaf nodes.

The SQL query depicted in Figure 11(b) (denoted by `SQL-02A`) is able to detect all inserted leaf nodes in newly inserted subtrees. The child nodes of an inserted node must also be inserted nodes. The SQL query `SQL-02A` basically retrieves the leaf child nodes of inserted internal nodes by using Definition 4.1.

All inserted leaf nodes in best matching subtrees are able to be detected by using the SQL query depicted in Figure 11(c) (denoted by `SQL-02B`). Let

r_1 and r_2 be two root nodes of best matching subtrees in first and second versions respectively. Let L_{r_1} and L_{r_2} be two sets of leaf nodes which are the child nodes of r_1 and r_2 respectively. Let $Y_{(r_1,r_2)}$ be a set of inserted leaf nodes in best matching subtrees rooted at r_1 and r_2. Intuitively, $Y_{(r_1,r_2)} = L_{r_2} - L_{r_1}$. This intuition is similar to the intuition of Definitions 4.3. The sub queries in lines $5-13$ and $15-23$ of `SQL-02B` are used to retrieve the leaf nodes that are the child nodes of r_2 and r_1 respectively. The `EXCEPT ALL` statement is used to find $y_i \in L_{r_2}$ where $\forall a_j \in L_{r_1}$ such that $(a_j \not\leftrightarrow y_i)$. In other words, this statement is used to find $Y_{(r_1,r_2)}$. Finally, we need to find other detailed information on inserted leaf nodes, such as the sibling orders and local orders, by joining $Y_{(r_1,r_2)}$ with L_{r_2} (lines 25-28).

The results of these SQL queries are stored in the `INS_LEAF` table as depicted in Figure 13(c). The semantics of attributes of the `INS_LEAF` table are depicted in Figures 12(d) and (e).

4.2.2 Deletion

Deletion of Internal Nodes

The deleted internal nodes can also be found by using the `AncestorInfo` and `Matching` tables. We use the similar attributes as for finding inserted internal nodes. The `DID1`, `DID2`, `Level`, `MinSO1`, and `MaxSO1` attributes of the `Matching` table are used to find deleted internal nodes. The SQL query depicted in Figure 11(a) can be used to detect a set of deleted internal nodes after slight modification. We replace "`MINSO2`" and "`MAXSO2`" in line 8 with "`MINSO1`" and "`MAXSO1`" respectively. We also replace the "*did2*" in line 7 to "*did1*". The modified SQL query depicted in Figure 11(a) is denoted by `SQL-03`. The SQL query `SQL-03` is able to detect all deleted internal nodes. The correctness of the SQL query `SQL-03` can be examined by following the similar intuitions as examining the correctness of the SQL query `SQL-01`. The result of this SQL query is stored in the `DEL_INT` table as depicted in Figure 13(b). The semantics of attributes of the `DEL_INT` table are depicted in Figures 12(d) and (e).

Deletion of Leaf Nodes

Similar to the inserted leaf nodes, the deleted leaf nodes are also classified into two categories: *deleted leaf nodes in deleted subtrees*, and *deleted leaf nodes in best matching subtrees*. We also use the SQL queries depicted in Figures 11(b) and (c) for detecting the deleted leaf nodes after slight modification. We replace "`INS_INT`" in line 5 in Figure 11(b) by "`DEL_INT`". We also replace the "*did2*" in line 7 in Figure 11(b) and in lines 11 and 25 in Figure 11(c) with "*did1*". The "*did1*" in line 21 in Figure 11(c) is replaced by "*did2*". We also replace "`MINSO2`" and "`MAXSO2`" in lines 13 and 26 in Figure 11(c) by "`MINSO1`" and "`MAXSO1`" respectively. The "`MINSO1`" and "`MAXSO1`" in line 23 in Figure 11(c) are replaced by "`MINSO2`" and "`MAXSO2`" respectively. The modified

```
1  SELECT DISTINCT
2    did1, did2, DL.PATH_ID, DL.LEVEL,
3    DL.LO AS LO1, IL.LO AS LO2,
4    DL.SO AS SO1, IL.SO AS SO2,
5    DL.VALUE AS V1, IL.VALUE AS V2
6  FROM DEL_LEAF AS DL,
7    INS_LEAF AS IL, MATCHING AS C
8  WHERE
9    DL.VALUE != IL.VALUE AND
10   DL.LO = IL.LO AND
11   DL.PATH_ID = IL.PATH_ID AND
12   DL.SO BETWEEN
        C.MINSO1 AND C.MAXSO1 AND
13   DL.LEVEL = (C.LEVEL+1) AND
14   IL.SO BETWEEN
        C.MINSO2 AND C.MAXSO2 AND
15   IL.LEVEL = (C.LEVEL+1) AND
16   IL.DID1 = did1 AND
17   IL.DID2 = did2 AND
18   DL.DID1 = did1 AND
19   DL.DID2 = did2
```

(a) Detecting Absolute Updated Leaf Nodes.

```
1  SELECT DISTINCT did1, did2, DL.PATH_ID, DL.LEVEL,
2    DL.LO AS LO1, IL.LO AS LO2, DL.SO AS SO1,
3    IL.SO AS SO2, DL.VALUE AS V1, IL.VALUE AS V2
4  FROM DEL_LEAF AS DL, INS_LEAF AS IL, MATCHING AS C
5  WHERE
6    DL.VALUE != IL.VALUE AND DL.LO != IL.LO AND
7    DL.PATH_ID = IL.PATH_ID AND
8    DL.SO BETWEEN C.MINSO1 AND C.MAXSO1 AND
9    DL.LEVEL = (C.LEVEL+1) AND
10   IL.SO BETWEEN C.MINSO2 AND C.MAXSO2 AND
11   IL.LEVEL = (C.LEVEL+1) AND
12   IL.DID1 = did1 AND IL.DID2 = did2 AND
13   DL.DID1 = did1 AND DL.DID2 = did2
```

(b) Detecting Relative Updated Leaf Nodes.

```
1  DELETE FROM DEL_LEAF
2  WHERE
3    DID1 = did1 AND DID2 = did2 AND
4    (PATH_ID, LO, SO, VALUE) IN
5      (SELECT PATH_ID, LO1, SO1, VALUE1
6       FROM UPD_LEAF
7       WHERE DID1 = did1 AND DID2 = did2)
```

(c) Delete Updated Leaf Nodes Detected as Deleted Leaf Nodes

DEL_INT (DID1, DID2, NAME, LEVEL, MINSO, MAXSO, LO)
INS_INT (DID1, DID2, NAME, LEVEL, MINSO, MAXSO, LO)
DEL_LEAF (DID1, DID2, PATH_ID, SO, LO, LEVEL, VALUE)
INS_LEAF (DID1, DID2, PATH_ID, SO, LO, LEVEL, VALUE)
UPD_LEAF (DID1, DID2, PATH_ID, LO1, LO2, SO1, SO2, LEVEL, VALUE1, VALUE2)
MOV_INT (DID1, DID2, NAME, LEVEL, LO1, LO2, MINSO1, MAXSO1, MINSO2, MAXSO2)
MOV_LEAF (DID1, DID2, LEVEL, LO1, LO2, PATH_ID, SO1, SO2, VALUE)

(d) Tables and Attributes

Attribute	Description
DID1	The document id of the first document
DID2	The document id of the second document
NAME	The internal node's name
LEVEL	The node's level
MINSO	The minimum sibling order of an internal node
MAXSO	The maximum sibling order of an internal node
SO	The sibling order of a leaf node
PATH_ID	The path id of a leaf node
VALUE	The leaf node's value
LO	The local order of a node
LO1	The local order of a node in the first version
LO2	The local order of a node in the second version
SO1	The sibling order of a leaf node in the first version
SO2	The sibling order of a leaf node in the second version
VALUE1	The old value of an updated node
VALUE2	The new value of an updated node

(e) Description of Attributes

Fig. 12. SQL Queries (2) and Table For Storing Delta.

SQL query depicted in Figures 11(b) and (c) are denoted by `SQL-04A` and `SQL-04B` respectively.

The correctness of `SQL-04A` and `SQL-04B` can be shown by following similar intuitions as showing the correctness of `SQL-02A` and `SQL-02B` respectively. The results of these SQL queries are stored in the `DEL_LEAF` table as shown in Figure 13(d). The semantics of attributes of the `DEL_LEAF` table are depicted in Figures 12(d) and (e).

4.2.3 Update

Update operations on leaf nodes can be classified into *absolute updates* and *relative updates*. In the *absolute update*, the node's position in DOM tree is not changed, but the value has changed. In the *relative update* operation, the absolute position as well as the value of the node has changed due to insert/delete/move operations on other nodes. We detect the updated leaf nodes by using the `INS_LEAF` and `DEL_LEAF` tables in which inserted and deleted leaf nodes are stored respectively. In addition, we also need to use the `Matching` table to guarantee that the updated leaf nodes are in best matching subtrees.

Detecting Absolute Updates

According to Definitions 4.6, we are able to detect absolute update operations by using the `DID1`, `DID2`, `Level`, `SiblingOrder`, `LocalOrder`, `Path_Id`, and

ID	DID1	DID2	Name	Level	MinSO	MaxSO	LO
114	1	2	staff	2	6	7	4
117	1	2	research	3	7	7	3

(a) Inserted Internal Nodes (INS_INT Table)

ID	DID1	DID2	Name	Level	MinSO	MaxSO	LO
3	1	2	staff	2	2	3	2
6	1	2	research	3	3	3	3

(b) Deleted Internal Nodes (DEL_INT Table)

ID	DID1	DID2	Path_ID	SO	LO	Level	Value
118	1	2	4	7	1	4	Semantic Web
115	1	2	2	6	1	3	Steve
116	1	2	3	6	2	3	Asst Prof
108	1	2	4	3	2	4	Information Retrieval
105	1	2	3	2	2	3	Prof

(c) Inserted Leaf Nodes (INS_LEAF Table)

ID	DID1	DID2	Path_ID	SO	LO	Level	Value
13	1	2	4	5	1	4	Data Mining
4	1	2	2	2	1	3	Smith
7	1	2	4	3	1	4	Web Mining
5	1	2	3	2	2	4	Assoc Prof
8	1	2	4	3	2	4	Multimedia Mining
17	1	2	3	6	2	3	Assoc Prof

(d) Deleted Leaf Nodes (DEL_LEAF Table)

ID1	ID2	DID1	DID2	Path_ID	LO1	LO2	SO1	SO2	Level	Value1	Value2
17	105	1	2	3	2	2	6	2	3	Assoc Prof	Prof

(e) Updated Leaf Nodes (UPD_LEAF Table)

Fig. 13. Delta.

`Value` attributes of the `INS_LEAF` and `DEL_LEAF` tables. For matching internal nodes in the `Matching` table, we use `DID1`, `DID2`, `Level`, `MinSO1`, `MaxSO1`, `MinSO2`, and `MaxSO2` attributes. The SQL query for detecting absolute update operations is shown in Figure 12(a) (denoted by `SQL-05A`).

The `SQL-05A` is able to find all absolute updated leaf nodes. Lines $12-15$ of `SQL-05A` are used to guarantee that pairs of deleted and inserted leaf nodes are the child nodes of the root nodes of best matching subtrees by following Definition 4.1. Each pair of leaf nodes must have the same local order (line 10), and the same path from the root nodes (line 11), but have different value (line 9). Recall that the updated leaf nodes are detected as pairs of deleted and inserted leaf nodes. Hence, the SQL query depicted in Figure 12(c) is used to delete the corresponding tuples of absolute updated leaf nodes detected as deleted leaf nodes in the `DEL_LEAF` table. We also need to delete the absolute updated nodes detected as inserted leaf nodes by using the SQL query depicted in Figure 12(c) (after slight modification). We replace "`DEL_LEAF`" in line 1 with "`INS_LEAF`". The "`LO1`", "`SO1`", and "`VALUE1`" in line 5 are replaced by "`LO2`", "`SO2`", and "`VALUE2`" respectively.

Detecting Relative Updates

We use the same attributes of the `INS_LEAF`, `DEL_LEAF`, and `Matching` tables as ones for detecting absolute updates. We also use the `UPD_LEAF` in which absolute update leaf nodes are stored in order to guarantee the leaf nodes that are already detected as absolute updates are not detected as relative updates. Based on Definition 4.7, the relative update operations can be detected by using the SQL query in Figure 12(b) (denoted by `SQL-05B`). Lines 8-11 are used to guarantee that the parent nodes of the updated leaf nodes are best matching internal nodes.

We observed that the result of the SQL query `SQL-05B` may not be correct for certain cases. Let us elaborate by using an example. Suppose we have two trees as depicted in Figure 14(a). The result of the SQL query depicted in Figure 12(b) is shown in Figure 14(b) (partial view only). We notice that nodes `B` with values "V2" and "V4" are detected as updated leaf nodes twice. This

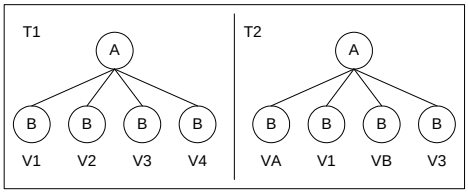
is because the SQL query depicted in Figure 12(b) only finds the leaf nodes which have the same paths, but different values and local orders. We use *updateCorrector* algorithm that is depicted in Figure 14(c) to correct the result. First, the algorithm determines the updated leaf nodes in the first version that are detected as updated leaf nodes more than once (line 2, Figure 14(c)) by using the SQL query *Q1* as depicted in Figure 14(d). Lines 17 and 18 in Figure 14(d) are used to retrieve only one row. The SQL query *Q1* returns R. Next, the algorithm deletes the *incorrect tuples* (line 3, Figure 14(c)) by using the SQL query in Figure 14(e). A tuple t is an *incorrect tuple* if one and only one of the following conditions is satisfied: 1) the `VALUE1` of tuple t is equal to `VALUE1` of R, 2) the `VALUE2` of tuple t is equal to `VALUE2` of R. We also do the same process for the updated leaf nodes in the second version that are detected as updated leaf nodes more than once. Note that the SQL query *Q2* is generated by slightly modifying the query in Figure 14(d). We replace the "`VALUE1`" in lines 7, 12, 13, and 20 in Figure 14(d) with '`VALUE2`". The *updateCorrector* algorithm results the `UPD_LEAF` table (without highlighted rows) as depicted in Figure 14(b). Note that we also need to delete the corresponding tuples of relative updated leaf nodes detected as pairs of deleted and inserted leaf nodes stored in the `DEL_LEAF` and `INS_LEAF` tables respectively. The detected updated leaf nodes are stored in the `UPD_LEAF` table as shown in Figure 13(e). The semantics of the attributes of the `UPD_LEAF` table are depicted in Figures 12(d) and (e).

4.2.4 Move Operations

In this section, we shall discuss how the move operations are detected. According to the discussion in Section 4, move operations are classified into *move among siblings* and *move to different parent nodes*. Let us elaborate further how to detect each type of move operations.

Move Among Siblings

The naive approach of detecting the movement of nodes among the siblings is to check whether or not the local order of a node has changed. However, this approach may lead to the detection of *non-optimal* deltas in certain situations. We illustrate this with a simple example. Suppose we have two versions of XML trees as depicted in Figure 15(a). Node $e2$ with value "New" is a newly inserted node. If we do not consider this newly inserted node during the move detection process, then we may detect that nodes $e2$ with values "C" and "D" are moved among their siblings since they have different local order values in the old and new versions. Hence, the detected delta consists of two move operations and an insert operation. However, the optimal delta should consist of only an insert operation. To overcome this problem, we need to simulate the insertions and deletions occurring under the same parent before detecting moved nodes.



(a) Example

DID1	DID2	LEVEL	SO1	SO2	PATH_ ID	LO1	LO2	VALUE 1	VALUE 2
..	..	..	..	..	..	2	1	V2	VA
..	..	..	..	..	..	2	3	V2	VB
..	..	..	..	..	..	4	1	V4	VA
..	..	..	..	..	..	4	3	V4	VB

(b) Updated Leaf Nodes

```
Input: Table UPD_LEAF, did1, did2
Output: Corrected Table UPD_LEAF

1  Algorithm updateCorrector {
2    while (result R of query Q1 is not empty){
3      correctUpdateTable(R);
4    }
5    while (result R of query Q2 is not empty){
6      correctUpdateTable(R);
7    }
8  }
```

(c) Algorithm *updateCorrector*

```
1  SELECT
2     U.SO1, U.SO2, U.PATH_ID,
3     U.VALUE1, U.VALUE2
4  FROM UPD_LEAF AS U,
5    (SELECT DID1, DID2, SO1, SO2,
6         PATH_ID, VALUE1, COUNT(VALUE1)
7     FROM UPD_LEAF
8     WHERE DID1 = did1 AND DID2 = did2
9     GROUP BY DID1, DID2, SO1, SO2,
10          PATH_ID, VALUE1
11    HAVING COUNT(VALUE1)>1) AS T
12 WHERE
13    U.DID1 = did1 AND U.DID2 = did2 AND
14    U.SO1 = T.SO1 AND U.SO2 = T.SO2 AND
15    U.PATH_ID = T.PATH_ID AND
16    U.VALUE1 = T.VALUE1
17 FETCH FIRST 1 ROWS ONLY
18 OPTIMIZE FOR 1 ROWS
```

(d) SQL Query (1)

```
1 DELETE FROM UPD_LEAF
2 WHERE
3   DID1 = did1 AND  DID2 = did2 AND
5   SO1 = R.SO1 AND SO2 = R.SO2 AND
6   PATH_ID = R.PATH_ID AND
7   ((VALUE1 = R.VALUE1 AND
8     VALUE2 != R.VALUE2) OR
9    (VALUE1 != R.VALUE1 AND
10    VALUE2 = R.VALUE2))
```

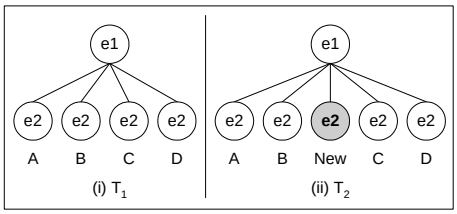
(e) SQL Query (2)

Fig. 14. Example of the Uncomplete Result in Update Operation.

Figure 15(b) depicts the algorithm for detecting the movement of nodes among their siblings. First, the algorithm generates a `moveList` table that initially consists of nodes that are in the matching subtrees. The SQL queries shown in Figures 15(c) and (d) are used to generate the `moveList` table. The SQL query depicted in Figure 15(c) joins the `Matching` table in order to guarantee that the moved leaf nodes are in the best matching subtrees. For example, the `moveList` table is depicted in Figure 16(a) (partial view only). The second step of the algorithm is to simulate the insertions and deletions by adjusting the local orders of the nodes affected by these operations. The adjustment of the local orders is based on the observations as in Section 4.1. For instance, the `moveList` table after the adjustment of the local orders is depicted in Figure 16(b) (partial view only). Finally, we delete the tuples in the `moveList` that have identical values of the `LO1` and `LO2` attributes. The `moveList` table after deleting the tuples that have identical values of the `LO1` and `LO2` attributes is shown in Figure 16(c). The moved internal nodes among their siblings are stored in the `MOV_INT` table. The moved leaf nodes among their siblings are stored in the `MOV_LEAF` table. The attributes of the `MOV_INT` and `MOV_LEAF` tables are depicted in Figures 12(d) and (e).

Move to Different Parent Nodes

A particular node that is moved to different parent node is detected as a pair of deletion and insertion. Hence, we are able to determine the nodes that are moved to different parent nodes by querying the `DEL_INT` and `INS_INT` tables (for moved internal nodes), and the `DEL_LEAF` and `INS_LEAF` tables (for moved leaf nodes). However, for the moved internal nodes, the subtrees that



(a) Example

```
Input: MATCHING, LEAFVALUE tables
Output: moveList table

1  generateMTable(T.minso1,T.maxso1,
       T.minso2, T.maxso2);
2  adjustLocalOrder();
3  DELETE FROM moveList WHERE LO1=LO2;
4  return moveList table
```

(b) Algorithm *findMoveAmongSibling*.

```
1  SELECT
2    did1, did2, P1.SIBLINGORDER,
3    0 AS MAXSO1, P2.SIBLINGORDER,
4    0 AS MAXSO2, P1.LEAFVALUE,
5    P1.LOCALORDER, P2.LOCALORDER, P1.LEVEL
6  FROM LEAFVALUE AS P1,
       LEAFVALUE AS P2, MATCHING AS C
7  WHERE
8    P1.DOC_ID = did1 AND
9    P2.DOC_ID = did2 AND
10   C.DID1 = did1 AND C.DID2 = did2 AND
11   C.SCORE < 1.000 AND
12   P1.PATH_ID = P2.PATH_ID AND
13   P1.LEVEL = (C.LEVEL+1) AND
14   P2.LEVEL = P1.LEVEL AND
15   P2.LEAFVALUE = P1.LEAFVALUE AND
16   P1.SIBLINGORDER BETWEEN
       C.MINSO1 AND C.MAXSO1 AND
17   P2.SIBLINGORDER BETWEEN
       C.MINSO2 AND C.MAXSO2
```

(c) First Query

```
1 SELECT
2  did1, did2, C.MINSO1, C.MAXSO1, C.MINSO2,
3  C.MAXSO2, '-' AS VALUE, C.LO1, C.LO2, C.LEVEL
4 FROM MATCHING AS C
5 WHERE DID1 = did1 AND DID2 = did2
```

(d) Second Query

Fig. 15. Move Among Siblings: Example, Algorithm, and SQL Queries.

DID1	DID2	MinSO1	MaxSO1	MinSO2	MaxSO2	Value	LO1	LO2	Level
1	2	1	-	1	-	Information System	1	1	2
1	2	4	5	4	5	-	3	3	2
1	2	6	7	2	3	-	4	2	2
1	2	...	...	...	...	...	...	...	...

(a) moveList Table: Intial state

DID1	DID2	MinSO1	MaxSO1	MinSO2	MaxSO2	Value	LO1	LO2	Level
1	2	1	-	1	-	Information System	1	1	2
1	2	4	5	4	5	-	2	3	2
1	2	6	7	2	3	-	3	2	2
1	2	...	...	...	...	-	...	...	...

(b) moveList Table: After applying insertions and deletions

DID1	DID2	MinSO1	MaxSO1	MinSO2	MaxSO2	Value	LO1	LO2	Level
1	2	4	5	4	5	-	2	3	2
1	2	6	7	2	3	-	3	2	2

(c) moveList Table: Final state

Fig. 16. Move Among Siblings: `moveList` Table.

are rooted at these moved internal nodes should be matching subtrees. This leads us to have a better quality of XDelta. Note that we only consider the movement of nodes to different parent nodes at the same level. The movement of nodes to different parent nodes at different level will be detected as pairs of deletion and insertion.

In the bottom-up approach, these moved internal nodes can be found by using the `DEL_INT`, `INS_INT`, and `TempMatching` tables. Recall that the `TempMatching` table has `Flag` attribute that is used to annotate the root nodes that are candidates to be the root nodes of moved subtrees. The possible moved subtrees have the `Flag` attribute equal to "1". This indicates that subtrees P (in the old version) and Q (in the new version) rooted at nodes p and q respectively are matching subtrees, but $parent(p) \not\equiv parent(q)$. $parent(p)$ and $parent(q)$ are detected as deleted and inserted internal nodes respectively. Hence, nodes p and q are also determined as deleted and inserted internal nodes. The SQL query in Figure 17(a) is used to detect the internal nodes that are moved to different parent nodes. The result of the SQL query is stored in the `MOV_INT` table. The next step is to find all the leaf nodes that are in moved subtrees. These leaf nodes can be found in the `DEL_LEAF` and `INS_LEAF` tables. The information on the leaf nodes that are in moved subtrees is stored in the `MOV_LEAF` table. The attributes of the `MOV_INT` and `MOV_LEAF` tables are depicted in Figures 12(d) and (e).

```
1  SELECT
2    D.DID1, D.DID2, T.LEVEL, I.NAME,
3    D.LOCALORDER, I.LOCALORDER,
4    D.MINSO, D.MAXSO, I.MINSO, I.MAXSO
5  FROM TEMPMATCHING AS T, INS_INT AS I,
6       DEL_INT AS D
7  WHERE
8    T.FLAG = 1 AND I.NAME = D.NAME AND
9    I.DID1 = did1 AND I.DID2 = did2 AND
10   D.DID1 = did1 AND D.DID2 = did2 AND
11   T.MINSO1 = D.MINSO AND
12   T.MAXSO1 = D.MAXSO AND
13   T.MINSO2 = I.MINSO AND
14   T.MAXSO2 = I.MAXSO AND
15   T.LEVEL = I.LEVEL AND
16   T.LEVEL = D.LEVEL
```

(a) Detecting Moved Internal Nodes

```
1  SELECT
2    D.DID1, I.DID2, D.LEVEL, D.PATH_ID, D.LOCALORDER,
3    I.LOCALORDER, D.SIBLINGORDER, I.SIBLINGORDER,
4    I.VALUE
5  FROM DEL_LEAF AS D, INS_LEAF AS I,
6       MATCHING AS C1, MATCHING AS C2
7  WHERE
8    D.DID1 = did1 AND D.DID2 = did2 AND
9    I.DID1 = did1 AND I.DID2 = did2 AND
10   C1.DID1 = did1 AND C1.DID2 = did2 AND
11   C2.DID1 = did1 AND C2.DID2 = did2 AND
12   D.PATH_ID = I.PATH_ID AND D.VALUE = I.VALUE AND
13   D.LEVEL = C1.LEVEL+1 AND I.LEVEL = C2.LEVEL+1 AND
14   C1.LEVEL = C2.LEVEL AND
15   D.MINSO >= C1.MINSO1 AND D.MAXSO <= C1.MAXSO1 AND
16   I.MINSO >= C2.MINSO2 AND I.MAXSO <= C2.MAXSO2
```

(b) Detecting Moved Leaf Nodes

Fig. 17. Move To Different Parent Nodes: SQL Queries.

In the top-down approach, finding the internal nodes that are moved to different parent nodes is a time-consuming process. Suppose we have two subtrees P (in the old version) and Q (in the new version) rooted at nodes p and q respectively. When $parent(p)$ has no matching subtree in the new version, all subtrees in the subtree rooted at $parent(p)$ (including subtree P) will not be compared to the subtrees in the new version. Similarly, when $parent(q)$ has no matching subtree in the old version, all subtrees in the subtree rooted at $parent(q)$ (including subtree Q) will not be compared to the subtrees in the old version. That is, we do not have information on the matching subtrees that are in the subtrees rooted at $parent(p)$ and $parent(q)$. This leads us to find the information on the matching subtrees that are in the subtrees rooted at $parent(p)$ and $parent(q)$. Hence, in the top-down approach, the moved internal nodes are detected as pairs of deleted and inserted internal nodes in order not to sacrifice the performance of the top-down approach.

The leaf nodes are also able to be moved to different parent nodes. Both approaches in XANDY are able to detect these moved leaf nodes. The SQL query depicted in Figure 17(b) is used to find the leaf nodes that are moved to different parent nodes. We also use the `Matching` table in order to make sure that the parent nodes of these moved leaf nodes are in both versions. Lines 13-16 are used to guarantee that the parent nodes of these moved leaf nodes are in both versions. Line 12 is used to ensure that these leaf nodes are matching leaf nodes. The result of the query is stored in the `MOV_LEAF` table.

5 Experimental Results

In this section, we examine the performance of XANDY approaches. The top-down and bottom-up approaches are implemented in Java. We ran the experiments on a Microsoft Windows 2000 Professional machine having Intel Pentium 4 1.7 GHz processor with 512 MB of memory. The database system we used was IBM DB2 UDB 8.1. We create two databases, one is for the top-down approach, and another is for the bottom-up approach. We specify the query workload to the Design Advisor, and the indexes on the relations are created based on the advice of The Design Advisor.

Dataset	Number of Nodes	Size (KB)
SIGMOD-01	331	13
SIGMOD-02	544	21
SIGMOD-03	890	34
SIGMOD-04	1,826	70
SIGMOD-05	2,718	104
SIGMOD-06	4,717	180

Dataset	Number of Nodes	Size (KB)
SIGMOD-07	8,794	337
SIGMOD-08	18,866	721
SIGMOD-09	37,725	1,444
SIGMOD-10	89,323	3,431
SIGMOD-11	172,754	6,635
SIGMOD-12	290,539	11,167

(a) SIGMOD Record

Dataset	Number of Nodes	Size (KB)
TCSD-01	1,239	51
TCSD-02	1,821	75
TCSD-03	3,062	129
TCSD-04	5,100	212

Dataset	Number of Nodes	Size (KB)
TCSD-05	16,526	686
TCSD-06	25,844	1,075
TCSD-07	41,803	1,745
TCSD-08	69,043	2,842

(b) Dictionary

Fig. 18. Dataset.

There are two synthetic data sets based on the SIGMOD Record DTD [1] (SIGMOD Data sets) and Oxford English Dictionary [2] (TCSD Data sets) [16]. SIGMOD data sets are represented the data-centric documents, and TCSD data sets are represented the text-centric documents. We generated the second version of each XML document by using our own change generator. We distributed the percentage changes equally for each type of changes. Figures 18(a) and (b) show the characteristics of the SIGMOD and TCSD data sets respectively. Note that we focus on the number of nodes in our data sets as the higher the number of nodes in a tree the database engine will involve more number of tuples for processing.

We also studied the performance of the state-of-the-art approaches. Unfortunately, despite our best efforts (including contacting the authors), we could not get the Java version of XyDiff [4]. Hence, we compared our approaches to the Java version of X-Diff[15] [3]. In addition, we also show the performance of the C version of XyDiff in order to know the performance of XyDiff in detecting the changes on our data sets. The C version of XyDiff was run in a Pentium 4 1.7 GHz processor with 512 MB of memory with Red Hat Linux 9 operating system.

5.1 Execution Time vs Number of Nodes

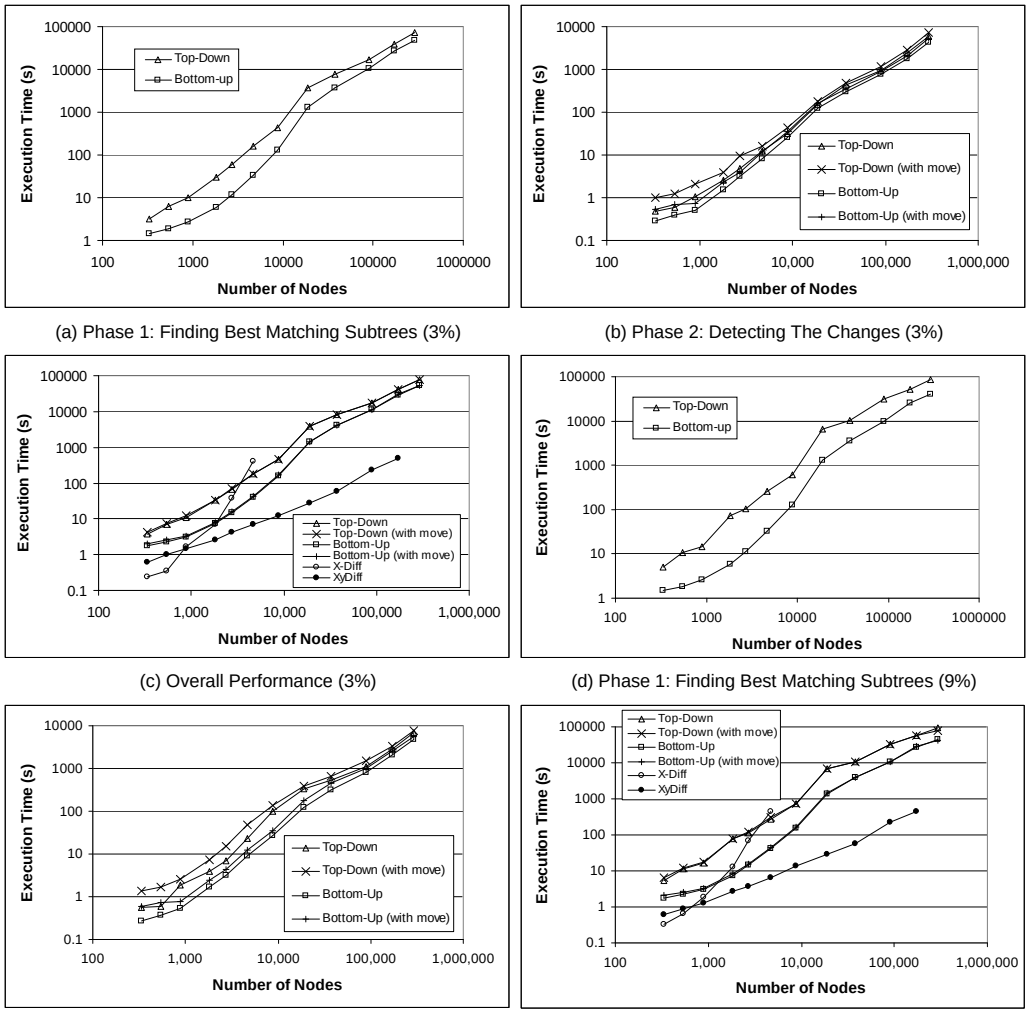
In these sets of experiments, we study the performance of our approaches for various sizes of XML documents. We use two data sets: SIGMOD data sets and TCSD data sets. We set the percentages of changes to "3%" and "9%" for the SIGMOD data sets, and to 3% for the TCSD data sets.

In the first set of experiments, we study the performance of the approaches by using SIGMOD data sets. Figures 19(a) and (d) depict the performance of the first phase ("Finding the best matching subtrees") of our approaches when the percentages of changes are set to "3%" and "9%" respectively. The bottom-up approach has a better performance than the top-down approach. In average, the performance of the bottom-up approach is about 5 times faster than the top-down approach. Figures 19(b) and (e) depict the performance of the second phase ("Detecting the changes") of our approaches when the percentages of changes are set to "3%" and "9%" respectively. We notice that the bottom-up

[1] http://www.acm.org/sigmod/record/xml/

[2] http://www.oed.com

[3] Downloaded from http://www.cs.wisc.edu/~yuanwang/xdiff.html



(a) Phase 1: Finding Best Matching Subtrees (3%)

(b) Phase 2: Detecting The Changes (3%)

(c) Overall Performance (3%)

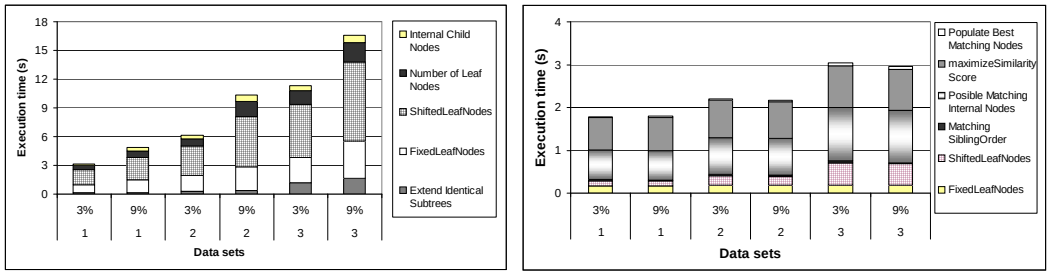
(d) Phase 1: Finding Best Matching Subtrees (9%)

(e) Phase 2: Detecting The Changes (9%)

(f) Overall Performance (9%)

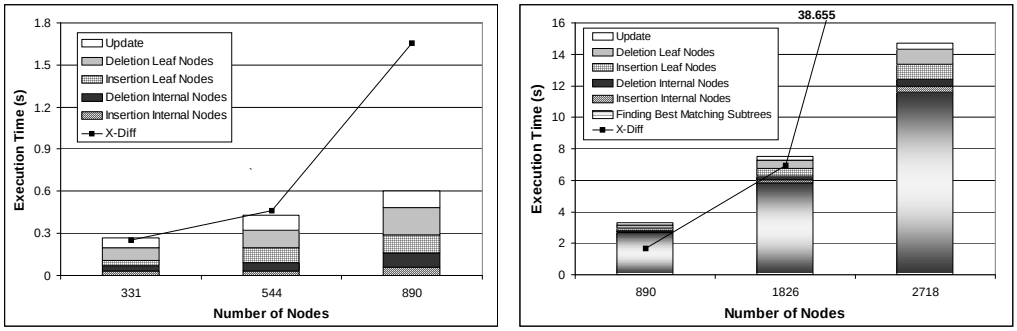
Fig. 19. Sigmod Data Sets - Execution Time vs Number of Nodes (Log Scale) .

approach is up to 3 times faster than the top-down approach. Note that the greedy approximation in the first phase of the top-down approach influence the performance in the second phase as the greedy approximation may match two subtrees that may not be best matching subtrees. Figures 19(c) and (f) depict the overall performance of our approaches, X-Diff, and XyDiff when the percentages of changes are set to "3%" and "9%" respectively. We notice that the bottom-up approach is faster than the top-down approach. X-Diff is faster than the bottom-up approach for the first four data sets when the percentage of changes is 3%. When 9% of the documents are changed, X-Diff is faster than the bottom-up approach for the first three data sets. Then, for the larger data sets, the bottom-up approach is up to 10 times faster than X-Diff. Compared to the top-down approach; X-Diff is faster than the top-down approach for the first five data sets when the percentages of changes are 3% and 9%. For the larger data sets, the top-down approach is up to 3 times faster than X-Diff. Observe that although XyDiff shows a better response time than our approach, it still suffers from scalability problem. XyDiff fails



(a) The Top-down Approach (b) The Bottom-up Approach

Fig. 20. Sigmod Data Sets - Execution Time vs Number of nodes (2).



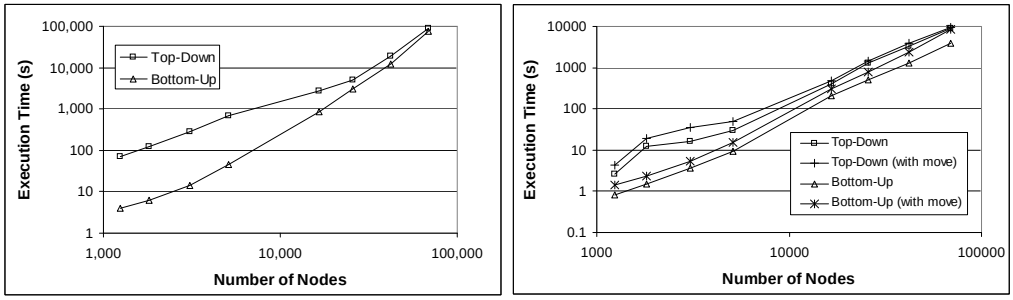
(a) Phase 2: Detecting Changes (b) Overall Performance

Fig. 21. Sigmod Data Sets - Execution Time vs Number of nodes (3).

to detect XDelta for "SIGMOD-12" as the process was killed by the kernel. Note that XyDiff is written in C and runs in Linux. We believe that the Java version of XyDiff will be much slower and less scalable than the C version and hence will adversely affect the response time and scalability further. Note that "SIGMOD-12" is almost two times larger than "SIGMOD-11".

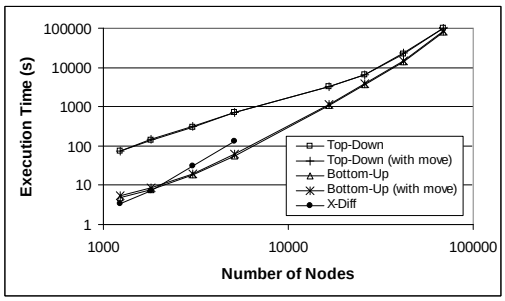
Figure 20(a) depicts the comparison between execution time of different SQL queries in the top-down approach for finding best matching subtrees. We notice that SQL query for calculating number of shifted leaf nodes (Figure 5(b)) takes up to 50% of the total execution time. Figure 20(b) depicts the comparison between execution time of different SQL queries in the bottom-up approach for finding best matching subtrees. The execution time of the algorithm for finding best matching configuration takes up to 42% of the total execution time. We also observe that the SQL queries for finding possible matching internal nodes takes up to 41% of the total execution time.

Figure 21(a) depicts the comparison between execution time for detecting each type of changes in the bottom-up approach and X-Diff. We noticed that most of the execution time of the second phase in the bottom-up approach are taken by the execution time for detecting deleted leaf nodes (around 31%) and inserted the leaf nodes (around 25%). Recall that we use two SQL queries as depicted in Figures 11(b) and (c) to detect the insertion/deletion of leaf nodes. The query cost of the SQL query in Figure 11(c) is higher than the one of the SQL query in Figure 11(b). The SQL query in Figure 11(b) only joins the LeafValue and INS_INT tables. The SQL query in Figure 11(c) contains three



(a) Phase 1: Finding Matching Subtrees (3%)

(b) Phase 2: Detecting The Changes (3%)



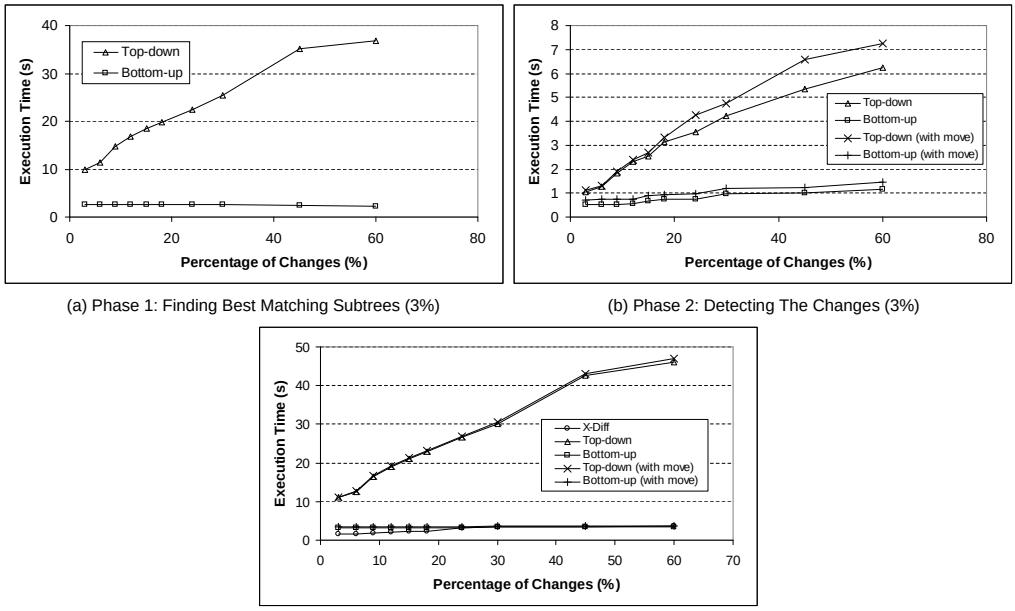
(c) Overall Performance (3%)

Fig. 22. TCSD Data Sets - Execution Time vs Number of Nodes.

queries. The second and first ones are in lines 5-13 and 15-23 respectively. These two queries are joined by using `EXCEPT ALL` statement. The result of the query in lines 5-23 is joined again with the `LeafValue` table in order to get the final result. It is obvious that the SQL query in Figure 11(c) is more expensive then the one in Figure 11(b). Figure 21(b) depicts the comparison between execution time for finding the best matching subtrees and detecting each type of changes in the bottom-up approach and X-Diff. We observed that the *Finding Best Matching Subtrees* phase (Phase 1) takes up to 81% of the overall execution time in average.

Based on the experiments, we study that the relational-based approach is more scalable than the memory-based approach. That is, X-Diff cannot detect the changes on "`SIGMOD-07`" and other larger data sets due to lack of memory, while our approaches are able to detect the changes to larger documents.

In this set of experiments, we study the performance of the approaches by using TCSD data sets. Figure 22(a) depicts the performance of the first phase ("Finding the best matching subtrees") of our approaches when the percentage of changes is set to "3%". We observe that the bottom-up approach is from 1.2 up to 20 times faster in average. Figure 22(b) depicts the performance of the second phase ("Detecting the changes") of our approaches when the percentage of changes is set to "3%". In average, the bottom-up approach is around 2.5 times faster than the top-down approach. Figure 22(c) depicts the overall performance of our approaches and X-Diff when the percentage of changes is set to "3%". We observe that the top-down approach is slower than X-Diff and the bottom-up approach. X-Diff is faster than the bottom-



(a) Phase 1: Finding Best Matching Subtrees (3%)

(b) Phase 2: Detecting The Changes (3%)

(c) Overall Performance (3%)

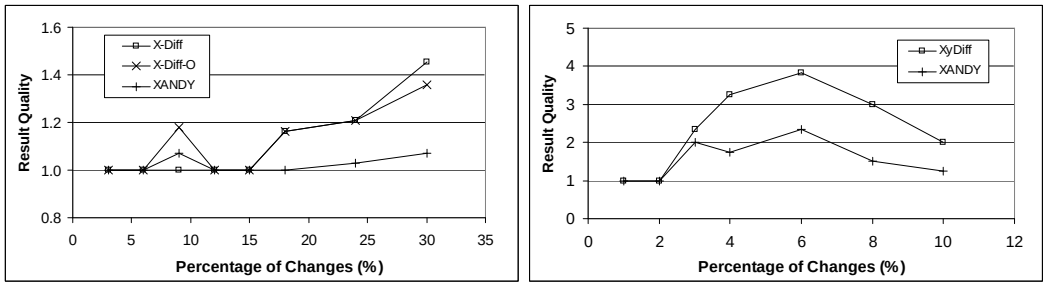
Fig. 23. Execution Time vs Percentage of Changes.

up approach when the number of nodes is less than 3000 nodes. When the number of nodes is greater than 3000 nodes, the bottom-up approach is up to 1.5 times faster than X-Diff. We also notice that X-Diff becomes less scalable. This is because text-centric documents contain actual text data. That is, we need more memory space to store the text data of text-centric documents. We also observed that the performances of detecting the changes on text-centric documents are worse than the performances of detecting the changes on data-centric documents.

5.2 Execution Time vs Percentage of Changes

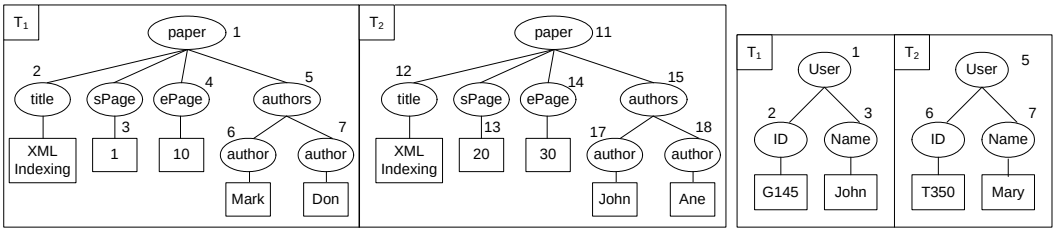
In the following set of experiments, we study the performance of our approaches for various percentages of changes by using "`SIGMOD-03`". We compare the performance of X-Diff, and our approaches.

Figure 23(a) depicts the performance of the first phase of both approaches. The top-down approach is negatively influenced by the percentages of changes. This means that when the documents are changed significantly, the performance of the top-down approach becomes worse. This is because there are more subtree comparisons. On the other hand, the bottom-up approach is positively influenced by the percentages of changes. When we increase the percentage of changes, the execution time of the bottom-up approach is faster. In this case, there is fewer matching leaf nodes. Hence, we shall find lesser number of possible matching subtrees when the algorithm moves upward in order to find best matching subtrees. That is, there will be lesser number of subtree comparisons. Figure 23(b) depicts the performance of the second phase of both approaches. We observe that the performances of both approaches are influ-



(a) Xandy vs X-Diff (b) Xandy vs XyDiff

Fig. 24. Result Quality.



(a) Example (1) (b) Example (2)

Fig. 25. Examples of Result Quality(1).

enced by the percentage of changes. The top-down approach is significantly influenced because of the greedy approximation in the first phase. Figure 23(c) depicts the overall performance of both approaches and X-Diff.

5.3 Result Quality

In the next set of experiments, the result quality of each approach is compared. The *result quality* is defined as the ratio between the number of edit operations in XDelta detected by an approach and the one in optimal XDelta. Note that we use the bottom-up approach of XANDY in these experiments as the result quality of the bottom-up approach is better than the result quality of the top-down approach. The top-down approach may not return optimal XDeltas in some cases due to the greedy approximation.

First, we compare the result quality of XANDY and of X-Diff. We used data set "`SIGMOD-02`", and constructed the second versions with various percentages of changes. We set the threshold Θ equal to "0.00". Figure 24(a) depicts the result quality of XANDY compared to of X-Diff. We study that the result quality of XANDY is comparable to X-Diff. Let us elaborate on this further. We notice that, in some cases, the result quality of X-Diff is better than of XANDY. Consider the example depicted in Figure 25(a). XANDY shall detect an XDeltas that consists of two update operations (nodes 3 and 4), a deletion of a subtree (subtree rooted at node 5), and an insertion of a subtree (subtree rooted at node 15). X-Diff shall detect an XDeltas that consists of a deletion of a subtree (subtree rooted at 1) and an insertion of a subtree (subtree rooted at node 11). We notice that, in some other cases, the result quality of XANDY is better than of X-Diff. Consider the example depicted in Figure 25(b). X-Diff

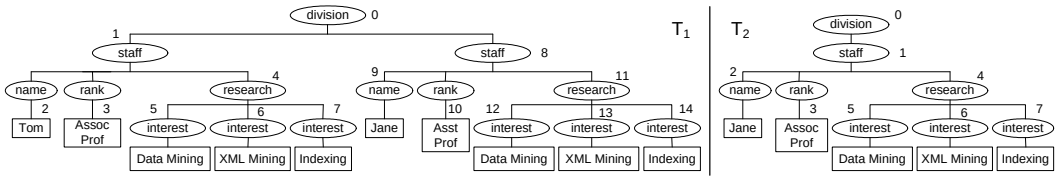


Fig. 26. Examples of Result Quality (2).

shall detect as two update operations (nodes 2 and 3). Note that updating the node "ID" is not semantically correct as ID number should not be updated. However, XANDY shall detect as a deletion of a subtree rooted at node 1 and an insertion of a subtree rooted at node 5. That is, X-Diff detects as a set of update operations, and XANDY detects as a pair of deletion and insertion of subtrees.

Next, we compare the result quality of XANDY and of XyDiff. We generate a set of XML documents based on the DTD of XML documents depicted in Figure 1. We generate the second versions with various percentages of changes. We set the threshold Θ equal to "0.00". We compare the results of XANDY and XyDiff with the optimal XDeltas. The ratios are depicted in Figure 24(b). We observed that, generally, XANDY has better result quality than XyDiff. Consider the example depicted in Figure 26. The delta detected by XANDY contains *delete*(1) and *update*(10, "Asst Prof", "Assoc Prof"). However, the delta generated by XyDiff contains *move*(9, 1, 2) [4] , *delete*(8), and *delete*(2). Note that the delta detected by XANDY is optimal delta.

6 Conclusions

The relational-based approach for ordered XML change detection system in this article is motivated by the scalability problem of existing memory-based approaches. We have shown that the relational approach is able to handle XML documents that are much larger than the ones detected by using main-memory approaches. We also report on the performance of two relational approaches in XANDY, the top-down and the bottom-up approaches, on two different kinds of data sets, the data-centric and the text-centric. We compare our approach to the published algorithm, X-Diff. We also show the performance of the C version of XyDiff in detecting the changes on our data sets. Our bottom-up approach has better performance compared to the top-down approach. The bottom-up approach is up to 4.5 times faster than the top-down approach. X-Diff outperforms our approaches for small XML data sets. For the larger XML data sets, the bottom-up approach is up to 10 times faster than X-Diff. We also notice that the type of data sets shall influence the performance and scalability of the approaches. The studies on the result quality have also been done in order to see the quality of deltas produced by our approaches. Our

[4] This operation means "move node 9 to the second child node of node 1"

bottom-up approach produces the deltas that are comparable to X-Diff and better than XyDiff.

References

[1] S. Chawathe, H. Garcia-Molina. Meaningful Change Detection in Structured Data. *In the Proceedings of the ACM SIGMOD*, 1997.

[2] S. Chawathe, A. Rajaraman, H. Garcia-Molina, J. Widom. Change Detection in Hierarchically Structured Information. *In the Proceedings of the ACM SIGMOD*, 1996.

[3] Y. Chen, S. Madria, S. S. Bhowmick. DiffXML: Change Detection in XML Data. *In the Proceedings of the 9th International Conference Database Systems for Advances Applications (DASFAA 2004)*, Jeju Island, Korea, 2004.

[4] G. Cobena, S. Abiteboul, A. Marian. Detecting Changes in XML Documents. *In the Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, San Jose, 2002.

[5] Curbera, D. A. Epstein. Fast Difference and Update of XML Documents. *XTech'99,* San Jose, 1999.

[6] D. Florescu, D. Kossmann. Storing and Querying XML Data Using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27-34, 1999.

[7] H. Jiang, H. Lu, W. Wang, J. Xu Yu. Path Materialization Revisited: An Efficient Storage Model for XML Data. *In Proceedings of Australasian Database Conference*, Melbourne, Australia, 2002.

[8] Erwin Leonardi, S. S. Bhowmick, T. S. Dharma, S. Madria. Detecting Content Changes on Ordered XML Documents Using Relational Databases. *In Proceedings of 15th International Conference Database and Expert Systems Applications (DEXA 2004)*, Zaragoza, Spain, 2004.

[9] M. Nicola, J. John. XML Parsing: A Threat to Database Performance. *In the Proceedings of the 12th ACM International Conference on Information and Knowledge Management (ACM CIKM 2003)*, New Orleans, USA, Nov 2003.

[10] H. Maruyama, K. Tamura, R. Uramoto. Digest Value for DOM (DOMHash). *IBM*, 2000.
`http://www.research.ibm.com/trl/projects/xml/xss4j/docs/rfc2803.html`

[11] Online Computer Library Center. Introduction to the Dewey Decimal Classification. `http://www.oclc.org/oclc/fp/about/about_the_ddc.htm` .

[12] S. Prakash, S. S. Bhowmick, S. Mardia. SUCXENT: An Efficient Path-based Approach to Store and Query XML Documents. *In Proceedings of 15th International Conference Database and Expert Systems Applications (DEXA 2004)*, Zaragoza, Spain, 2004.

[13] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. *In Proceedings of 25th International Conference on Very Large Data Bases (VLDB 1999)*, Edinburgh, Scotland, UK, 1999.

[14] T. F. Smith, M. S. Waterman. Identification of Common Molecular Subsequences. *In the Proceedings of Journal Molecular Biology* 147:195-197, 1981.

[15] Y. Wang, D. J. DeWitt, J. Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. *In the Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, Bangalore, 2003.

[16] B. B. Yao, M. T. Özsu, N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. *In the Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*, Boston, USA, 2004.

[17] M. Yoshikawa, T. Amagasa, T. Shimura, S. Uemura. XRel: A Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *In the Proceedings of ACM Transactions on Internet Technology*, 1(1):110-141, 2001.