# OXONE: A Scalable Solution for Detecting Superior Quality Deltas on Ordered Large XML Documents

Erwin Leonardi Sourav S. Bhowmick

School of Computer Engineering, Nanyang Technological University, Singapore
{pk909134, assourav}@ntu.edu.sg

**Abstract.** Recently, a number of relational-based approaches for detecting the changes to XML data have been proposed to address the scalability problem of main memory-based approaches (e.g., X-Diff, XyDiff). These approaches store the XML documents in the relational database and issue SQL queries (whenever appropriate) to detect the changes. In this paper, we propose a relational-based *ordered* XML change detection technique (called OXONE) that uses a *schema-conscious* approach as the underlying storage strategy for XML data. Previous efforts have focused on detecting changes to ordered XML in an *schema-oblivious* storage environment. Although the schema-oblivious approach produces better *result quality* compared to XyDiff (a main memory-based ordered XML change detection approach), its performance degrade with increase in data size and is slower than XyDiff for smaller data set. We propose a technique to overcome these limitations. Our experimental results show that OXONE is up to 22 times faster and more scalable than the relational-based schema-oblivious approach. The performances of OXONE and XyDiff (C version) are comparable. However, more importantly, our approach is more scalable compared to XyDiff for larger datasets and has much superior the result quality of deltas than XyDiff.

## 1 Introduction

Detecting changes to XML data is an important research problem. Recently, a number of main memory-based techniques for detecting the changes to XML data has been proposed. XyDiff [1] is an approach for detecting the changes to *ordered* XML documents. In an *ordered* XML, both the parent-child relationship and the left-to-right order among siblings are important. Wang et al. proposed X-Diff [8] for computing the changes to *unordered* XML documents. In *unordered* XML, the parent-child relationship is significant, while the left-to-right order among siblings is not important. All these algorithms suffer from scalability problem as they fail to detect changes to large XML documents due to lack of main memory.

In [3, 4], we have addressed this scalability problem in the context of unordered XML documents by leveraging on the relational technology. In this approach, given the old and new versions of an XML document, we store both documents in a relational database. Next, we issue a set of SQL queries (wherever appropriate) to detect the changes. Efficient and accurate change detection in such a relational environment is largely determined by the underlying storage structure. Particularly, there are two major approaches for storing XML documents in a relational database [7]. In *schema-conscious approach*, a relational schema is created based on the DTD/schema of the

XML documents. In the *schema-oblivious approach*, a fixed schema used to store XML documents is maintained. The basic idea is to capture the tree structure of an XML document. This approach does not require existence of an XML schema/DTD. In [2, 3], we have used schema-oblivious approach to detect changes to both *ordered* and *unordered* XML documents. Whereas, in [4], we proposed a schema-conscious driven approach for detecting changes to *unordered* XML data.

In this paper, we present a relational-based approach, called OXONE[1] (schema-c**O**nscious **X**ML-enabled **O**rdered cha**N**ge d**E**tection), for detecting the changes to *ordered* XML data using a *schema-conscious approach* (Shared-Inlining [6] in our case). Our effort is motivated by the following observations. First, a growing body of work suggests that schema-conscious approaches perform better than majority of the schema-oblivious approaches as far as XML query processing is concerned [7]. Second, our recent effort for detecting changes to *unordered* XML data in [4] using schema-conscious approach shows encouraging results. In particular, we have shown that the schema-conscious driven approach is significantly more scalable and faster than not only X-Diff [8] but also relational-based schema-oblivious approach such as XANDY [3].

At this point one may question the justification of this work as we have already explored the feasibility of using schema-conscious storage approach for detecting changes to XML data. However, the work reported in this paper is important for the following reasons. First, in [4] we have focused on change detection to *unordered* XML whereas in this paper we focus on *ordered* XML data. Although some of the SQL queries introduced in [4] can be used for detecting changes to ordered XML with minor modifications, as we shall see later, the very nature of ordered XML pose new challenges. For instance, unlike unordered change detection, ordered XML change detection has additional *move* operation that needs to be detected accurately. Second, the characteristics of schema-conscious approach raise certain challenges. Unlike schema-oblivious approaches, the underlying relational schema is DTD-dependent. Consequently, the challenge is to create a general framework for change detection so that the framework is independent of the structural heterogeneity of various XML documents. Third, it has been shown in [8] that XyDiff is significantly faster than X-Diff. However, the *result quality* of XyDiff is significantly poorer compared to X-Diff [8]. In [2, 3], we have shown that it is possible to generate superior quality deltas for both ordered and unordered XML change detection problem using relational-based approach. However, due to the underlying storage strategy, the relational-based approach in [2] is significantly slower than XyDiff and does not scale well with large data. *Consequently, is it possible to design a relational-based ordered XML change detection system that is more scalable and generates superior quality results, yet have response time which is at least comparable to XyDiff if not better?* In this paper, we propose OXONE to address these challenges.

In our approach, we first store two versions of an XML document, namely, $T_1$ and $T_2$, in a relational database whose underlying storage scheme is based on *modified* Shared-Inlining approach [6]. Then, OXONE can be used to detect the changes to $T_1$ and $T_2$ in a bottom-up fashion. Our approach consists of two phases: *finding best matching subtrees* phase and the *change detection* phase. The objective of the first phase is to find the most similar subtrees in $T_1$ and $T_2$. In order to find the most similar subtrees, we need to match subtrees in $T_1$ to ones in $T_2$. Note that a subtree in $T_1$ can

[1] pronounced as "ozone".

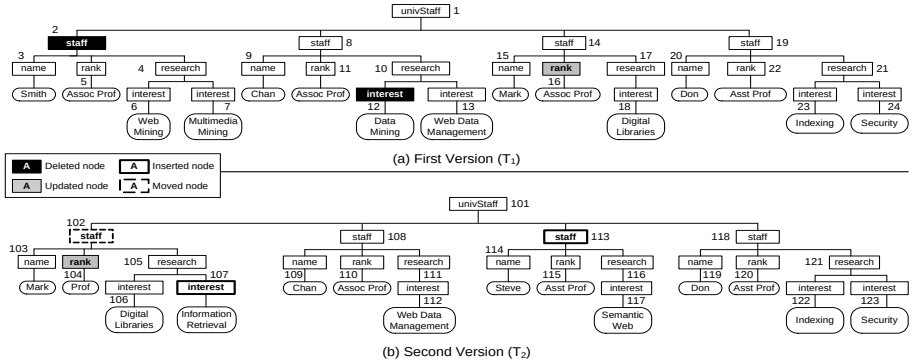



**Fig. 1.** Two versions of XML documents.

be matched to more than one subtree in $T_2$, and vice versa. In addition, we need to measure the similarity of each matching by calculating the *similarity score*. The most similar matching subtrees are called *best matching subtrees*. In our approach, we issue SQL queries (whenever appropriate) to find the best matching subtrees. We shall elaborate on this phase in Section 3. Having determined the best matching subtrees between $T_1$ and $T_2$, in the second phase OXONE issues SQL queries (whenever appropriate) to detect different types of changes. The types of changes that can be detected by OXONE are similar to the one in [1]. The detected changes are stored in several relations. We shall elaborate on this phase in Section 4.

We have implemented the prototype of OXONE on top Microsoft SQL Server 2000 using Java. We compared OXONE to XANDY–O [2], a published schema-oblivious ordered XML change detection system, and XyDiff [1]. Our results show that OXONE has comparable response time with XyDiff for large XML documents. However, it is more scalable and has superior *result quality* compared to XyDiff. Particularly, XyDiff fails to detect changes to XML documents containing around 356,000 nodes or more. Also, OXONE outperforms XANDY–O by up to 22 times and is more scalable. In addition, for larger data sets, OXONE is up to 44 times faster than X-Diff [8]. X-Diff is unable to detect the changes on XML documents that have more than 5000 nodes due to lack of main memory. We shall elaborate on the experimental results in Section 5. Note that the framework discussed in this paper is only for XML documents whose schemas do not contain recursive elements.

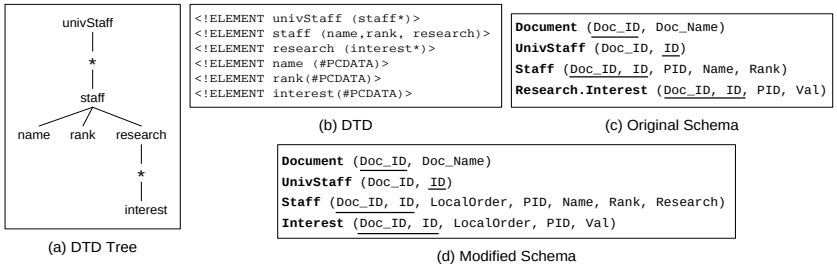



**Fig. 2.** DTD Tree, DTD, and Relational Schema.

## 2 Background

In this section, we first define some terms that we shall use subsequently to facilitate exposition. Then, we discuss how the Shared-Inlining schema is modified to support ordered XML change detection. We use the two versions of XML document in Figure 1 as running example throughout the paper.

### 2.1 Terminology

Let $T$ be a tree representation of an XML document $D$. The root node of $T$ is denoted by $root(T)$. Let $\mathcal{L}(T) = \{\ell_1, \ell_2, ..., \ell_n\}$ be a set of leaf nodes in XML tree $T$. The textual content of a leaf node $\ell$ is denoted by $value(\ell)$. A set of internal nodes in $T$ is denoted by $\mathcal{I}(T)$, and $i$ denotes an internal node, where $i \in \mathcal{I}$. The name and level of node $n$ are denoted by $name(n)$ and $level(n)$, respectively. Then, $path(n)$ denotes the path from $root(T)$ to node $n$. The parent node, child node, and ancestor node of node $n$ are denoted as $parent(n)$, $child(n)$, and $ancestor(n)$, respectively. In ordered XML, the left-to-right position of a node among its siblings is significant. Hence, $pos(n)$ denotes the left-to-right position of node $n$ among its siblings if $D$ is an *ordered* XML. Note that we use $T_1$ and $T_2$ as depicted in Figures 1(a) and 1(b), respectively, as our running example in the later discussion.

Let $\ell_{1_x} \in \mathcal{L}(T_1)$ and $\ell_{2_y} \in \mathcal{L}(T_2)$ be two leaf nodes in the first and second versions of an XML tree respectively. Then, $\ell_{1_x}$ and $\ell_{2_y}$ are *matching leaf nodes* (denoted as $\ell_{1_x} \leftrightarrow \ell_{2_y}$) if *name*($\ell_{1_x}$) = *name*($\ell_{2_y}$), *level*($\ell_{1_x}$) = *level*($\ell_{2_y}$), *path*($\ell_{1_x}$) = *path*($\ell_{2_y}$), and *value*($\ell_{1_x}$) = *value*($\ell_{2_y}$). For example, leaf nodes $\ell_{13}$ and $\ell_{112}$ are matching leaf nodes ($\ell_{13} \leftrightarrow \ell_{112}$) because they satisfy the above conditions. Note that a leaf node in $T_1$ can be matched to more than one leaf node in $T_2$, and vice versa. Leaf node $\ell_{110}$ in $T_2$ can be matched to node $\ell_4$, $\ell_{10}$, and $\ell_{16}$ in $T_1$ as they satisfy the above conditions. Note that if $\ell_1$ and $\ell_2$ are not matching leaf nodes, then they are denoted by $\ell_1 \not\leftrightarrow \ell_2$.

We classify the matching leaf nodes into two types, namely, *fixed matching leaf nodes* and *shifted matching leaf nodes*. This classification is important in the context of ordered change detection as if the left-to-right position among siblings of a node is changed, then it is possible that this node is moved among its siblings. Formally, let $\ell_1 \leftrightarrow \ell_2$. If $pos(\ell_1) = pos(\ell_2)$, then $\ell_1$ and $\ell_2$ are *fixed matching leaf nodes*. Otherwise, they are *shifted matching leaf nodes*. For example, leaf nodes $\ell_{18}$ and $\ell_{106}$ are fixed matching leaf node as $\ell_{18} \leftrightarrow \ell_{106}$ and $pos(\ell_{18}) = pos(\ell_{106})$. Leaf nodes $\ell_{13}$ and $\ell_{112}$ are shifted matching leaf node as $\ell_{13} \leftrightarrow \ell_{112}$ and $pos(\ell_{13}) \neq pos(\ell_{112})$.

Next, we define the notion of *matching leaf node groups*. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two sets of leaf nodes whose parent nodes are $i_1$ and $i_2$, respectively, where $i_1 \in \mathcal{I}(T_1)$ and $i_2 \in \mathcal{I}(T_2)$. Then, $\mathcal{G}_1$ and $\mathcal{G}_2$ are *matching leaf node groups* (denoted as $\mathcal{G}_1 \Leftrightarrow \mathcal{G}_2$) iff $\exists \ell_x \exists \ell_y$ such that $\ell_x \leftrightarrow \ell_y$, where $\ell_x \in \mathcal{G}_1$ and $\ell_y \in \mathcal{G}_2$. For example, suppose $\mathcal{G}_{17} = \{\ell_{18}\}$ and $\mathcal{G}_{105} = \{\ell_{106}, \ell_{107}\}$ are two sets of leaf nodes in $T_1$ and $T_2$ whose parent nodes are nodes 17 and 105, respectively. We observe that $\mathcal{G}_{17} \Leftrightarrow \mathcal{G}_{105}$ as $\ell_{18} \leftrightarrow \ell_{106}$, $\ell_{18} \in \mathcal{G}_{17}$, and $\ell_{106} \in \mathcal{G}_{105}$.

Next, we define *matching subtrees*. The root nodes of two matching subtrees are called *matching internal nodes*. From a set of matching subtrees, we determine the most similar subtrees to be *best matching subtrees*. Similar to X-Diff [8] and XyDiff [1], we

only match two subtrees at the same level. Formally, the *matching subtrees* are defined as follows. Let $t_1$ and $t_2$ be two subtrees rooted at nodes $i_1 \in \mathcal{I}(T_1)$ and $i_2 \in \mathcal{I}(T_2)$, respectively. Then, $t_1$ and $t_2$ are *matching subtrees* (denoted by $t_1 \doteq t_2$) if $name(i_1) = name(i_2)$, $level(i_1) = level(i_2)$, $path(i_1) = path(i_2)$, and $\exists p\ \exists q$ such that $p \leftrightarrow q$, where $i_1 = ancestor(p)$, $i_2 = ancestor(q)$, $p \in \mathcal{L}(T_1)$, and $q \in \mathcal{L}(T_2)$. For instance, the subtrees rooted at node 8 in $T_1$ and node 108 in $T_2$ are matching subtrees ($t_8 \doteq t_{108}$) as they have three matching leaf nodes ($\ell_9 \leftrightarrow \ell_{109}$, $\ell_{10} \leftrightarrow \ell_{110}$, and $\ell_{13} \leftrightarrow \ell_{112}$). If $t_1$ and $t_2$ are not matching subtrees, then they are denoted by $t_1 \neq t_2$. We use the terms of *matching subtrees* and *matching internal nodes* interchangeably.

Having found a set of matching subtrees, we need to measure the degree of similarity between two matching subtrees. We now define a metric called *similarity score* to measure how similar two subtrees are. The similarity score $\Re$ of two subtrees $t_1$ and $t_2$ that are in $T_1$ and $T_2$, respectively, 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, $|A|$ and $|B|$ are numbers of nodes of fixed and shifted matching leaf nodes in $t_1$ and $t_2$, respectively and $A \cap B = \emptyset$. For example, the similarity score of $t_8$ in $T_1$ and $t_{108}$ in $T_2$ is $\Re(t_8, t_{108}) = 0.714$. The value of similarity score is between 0 and 1. Two subtrees are more similar if the similarity score is higher. Based on the similarity score, we classify the subtrees into two types as follows. If $0 < \Re(t_1, t_2) \leq 1$, then the subtrees are *matching subtrees* and they have at least one matching leaf node. Otherwise, the subtrees are *unmatching subtrees* and they do not have matching leaf nodes ($\Re(t_1, t_2) = 0$).

Next, based on the above concepts, the *best matching subtrees* are formally defined as follows. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two sets of subtrees that are in $T_1$ and $T_2$, respectively. Let $t \in \mathcal{T}_1$ be a subtree and $P \subseteq \mathcal{T}_2$ be a set of subtrees. Also $t$ and $t_i \in P$ are matching subtrees $\forall\ 0 < i \leq |P|$. Then, $t$ and $t_i$ are *best matching subtrees* (denoted by $t \doteqdot t_i$) iff $(\Re(t, t_i) > \Re(t, t_j))\ \forall\ 0 < j \leq |P|$ and $i \neq j$. For example, subtree $t_{14}$ can be matched to subtrees $t_{102}$ and $t_{108}$. Observe that $\Re(t_{14}, t_{102}) = 0.571$ and $\Re(t_{14}, t_{108}) = 0.333$. Consequently, subtrees $t_{14}$ and $t_{102}$ are best matching subtrees ($t_{14} \doteqdot t_{102}$). Note that if $t_1$ and $t_2$ are not best matching subtrees, then they are denoted by $t_1 \not\doteqdot t_2$.

### 2.2 Extension of Shared-Inlining Approach

Recall that the OXONE approach is based on the Shared-Inlining storage strategy. For instance, given a DTD depicted in Figure 2(b), Shared-Inlining approach generates a relational schema as depicted in Figure 2(c). In [6], Shared-Inlining approach does not explicitly store the local order of nodes which is important in ordered XML documents. As this information is critical for our change detection process, we need to extend the relational schema generated by Shared-Inlining approach.

Before we discuss the extensions, let us present some notations that will be used in later discussion. Given a DTD tree $\mathcal{H}_U$ that is tree representation of DTD $U$, the nodes in $\mathcal{H}_U$ are classified as *inlined* and *non-inlined* nodes. An *inlined node* is one that is not below "$*$" or "$+$" node. There are two types of inlined nodes, namely, *inlined leaf nodes* (denoted by $\mathbb{I}_\ell$) and *inlined internal nodes* (denoted by $\mathbb{I}_i$). For example, consider a DTD tree as depicted in Figure 2(a). An inlined node will be stored as an attribute in the relation of its parent nodes. For example, the parent nodes of node *name* and *research* are node *staff*. The information on nodes *name* and *research* is stored in the

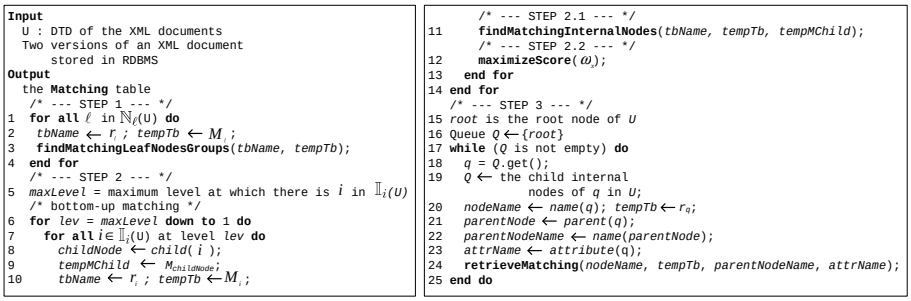



**Fig. 3.** The *findBestMatchingSubtrees* Algorithm.

*Name* attribute in the `Staff` table (Figure 2(c)). A *non-inlined* node is one that is below "$*$" or "$+$" node. There are also two types of non-inlined nodes, namely, *non-inlined leaf nodes* (denoted by $\mathbb{N}_\ell$) and *non-inlined internal nodes* (denoted by $\mathbb{N}_i$). An non-inlined node will be stored in a separate relation. For example, nodes *interest* and *staff* are a non-inlined leaf node and a non-inlined internal node whose information are stored in the `Interest` and `Staff` tables (Figure 2(c)), respectively.

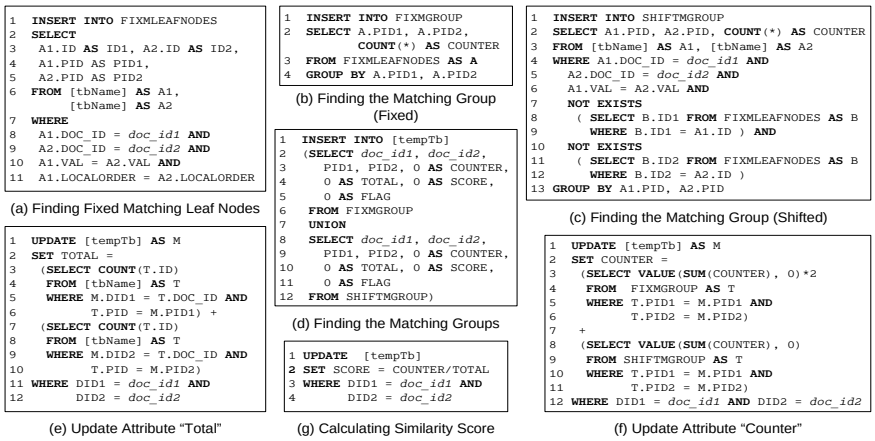
Let us now elaborate on the extensions of relational schema generated by Shared-Inlining approach. We add the *LocalOrder* attribute to the corresponding relations of non-inlined nodes. We store the information on inlined internal nodes as a `BOOLEAN` attribute (e.g., *research* attribute) in its parent relation. The extended relational schema is depicted in Figure 2(d). The *PID* in the figure refers to the parent node id.

## 3 Finding Best Matching Subtrees Phase

The *findBestMatchingSubtrees* algorithm is depicted in Figure 3. Note that "`[param]`" in the SQL queries (Figures 4 and 7) used in the later discussion will be replaced the parameter *param* defined in the algorithm. Also, due to space constraints, in our subsequent discussions we will not elaborate on queries and algorithms that are similar to the ones discussed in [4]. Rather, we shall highlight the differences (if any).

### 3.1 Finding Matching Leaf Nodes Groups Phase

The *findMatchingLeafNodesGroups* algorithm for finding matching leaf nodes groups works as follows. First, the *findMatchingLeafNodesGroups* algorithm determines the *fixed matching leaf nodes* by using the SQL query in Figure 4(a). Lines 10–11 are used to ensure that fixed matching leaf nodes have the same values and local orders. Next, we determine the *matching leaf nodes groups* from a set of fixed matching leaf nodes. The SQL query in Figure 4(b) is used to determine matching leaf nodes groups from a set of fixed matching leaf nodes. The idea behind this SQL query is to group the fixed matching leaf nodes by their *PID1* and *PID2* attributes (line 4, Figure 4(b)). Observe that the *PID1* and *PID2* attributes store the parent node id of fixed matching leaf nodes in the old and new versions, respectively. The next step is to determine *matching leaf nodes groups* from *shifted matching leaf nodes*. We use the SQL query in Figure 4(c).



```
1  INSERT INTO FIXMLEAFNODES
2  SELECT
3   A1.ID AS ID1, A2.ID AS ID2,
4   A1.PID AS PID1,
5   A2.PID AS PID2
6  FROM [tbName] AS A1,
        [tbName] AS A2
7  WHERE
8   A1.DOC_ID = doc_id1 AND
9   A2.DOC_ID = doc_id2 AND
10  A1.VAL = A2.VAL AND
11  A1.LOCALORDER = A2.LOCALORDER
```

(a) Finding Fixed Matching Leaf Nodes

```
1  INSERT INTO FIXMGROUP
2  SELECT A.PID1, A.PID2,
          COUNT(*) AS COUNTER
3  FROM FIXMLEAFNODES AS A
4  GROUP BY A.PID1, A.PID2
```

(b) Finding the Matching Group (Fixed)

```
1  INSERT INTO SHIFTMGROUP
2  SELECT A1.PID, A2.PID, COUNT(*) AS COUNTER
3  FROM [tbName] AS A1, [tbName] AS A2
4  WHERE A1.DOC_ID = doc_id1 AND
5    A2.DOC_ID = doc_id2 AND
6    A1.VAL = A2.VAL AND
7    NOT EXISTS
8     ( SELECT B.ID1 FROM FIXMLEAFNODES AS B
9       WHERE B.ID1 = A1.ID ) AND
10   NOT EXISTS
11    ( SELECT B.ID2 FROM FIXMLEAFNODES AS B
12      WHERE B.ID2 = A2.ID )
13 GROUP BY A1.PID, A2.PID
```

(c) Finding the Matching Group (Shifted)

```
1  UPDATE [tempTb] AS M
2  SET TOTAL =
3   (SELECT COUNT(T.ID)
4    FROM [tbName] AS T
5    WHERE M.DID1 = T.DOC_ID AND
6          T.PID = M.PID1) +
7   (SELECT COUNT(T.ID)
8    FROM [tbName] AS T
9    WHERE M.DID2 = T.DOC_ID AND
10         T.PID = M.PID2)
11 WHERE DID1 = doc_id1 AND
12       DID2 = doc_id2
```

(e) Update Attribute "Total"

```
1  INSERT INTO [tempTb]
2  (SELECT doc_id1, doc_id2,
3     PID1, PID2, 0 AS COUNTER,
4     0 AS TOTAL, 0 AS SCORE,
5     0 AS FLAG
6   FROM FIXMGROUP
7   UNION
8   SELECT doc_id1, doc_id2,
9     PID1, PID2, 0 AS COUNTER,
10    0 AS TOTAL, 0 AS SCORE,
11    0 AS FLAG
12  FROM SHIFTMGROUP)
```

(d) Finding the Matching Groups

```
1 UPDATE  [tempTb]
2 SET SCORE = COUNTER/TOTAL
3 WHERE DID1 = doc_id1 AND
4       DID2 = doc_id2
```

(g) Calculating Similarity Score

```
1  UPDATE [tempTb] AS M
2  SET COUNTER =
3   (SELECT VALUE(SUM(COUNTER), 0)*2
4    FROM  FIXMGROUP AS T
5    WHERE T.PID1 = M.PID1 AND
6          T.PID2 = M.PID2)
7   +
8   (SELECT VALUE(SUM(COUNTER), 0)
9    FROM SHIFTMGROUP AS T
10   WHERE T.PID1 = M.PID1 AND
11         T.PID2 = M.PID2)
12 WHERE DID1 = doc_id1 AND DID2 = doc_id2
```

(f) Update Attribute "Counter"

**Fig. 4.** SQL Queries for Finding Matching Leaf Nodes.

| DID1 | DID2 | PID1 | PID2 | Counter | Total | Score |
|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 1 | 3 | 0.333 |
| 1 | 2 | 3 | 1 | 2 | 3 | 0.666 |
| 1 | 2 | 4 | 4 | 4 | 4 | 1.000 |

(a) M_Interest Table

| DID1 | DID2 | ID1 | ID2 | LO1 | LO2 | PID1 | PID2 | Counter | Total | Score | Flag |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 1 | 1 | 1 | *null* | *null* | 17 | 29 | 0.586 | 0 |

(c) M_univStaff Table

| DID1 | DID2 | ID1 | ID2 | LO1 | LO2 | PID1 | PID2 | Counter | Total | Score | Flag |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 2 | 1 | 2 | 1 | 1 | 2 | 8 | 0.250 | 0 |
| 1 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 5 | 7 | 0.714 | 0 |
| 1 | 2 | 3 | 1 | 3 | 1 | 1 | 1 | 4 | 7 | 0.571 | 0 |
| 1 | 2 | 3 | 2 | 3 | 2 | 1 | 1 | 2 | 6 | 0.333 | 0 |
| 1 | 2 | 4 | 3 | 4 | 3 | 1 | 1 | 2 | 7 | 0.286 | 0 |
| 1 | 2 | 4 | 4 | 4 | 4 | 1 | 1 | 8 | 8 | 1.000 | 0 |

(b) M_Staff Table

| DID1 | DID2 | ID1 | ID2 | LO1 | LO2 | PID1 | PID2 | Score | Name |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 1 | 1 | 1 | *null* | *null* | 0.586 | univStaff |
| 1 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 0.714 | staff |
| 1 | 2 | 3 | 1 | 3 | 1 | 1 | 1 | 0.571 | staff |
| 1 | 2 | 4 | 4 | 4 | 4 | 1 | 1 | 1.000 | staff |
| 1 | 2 | *null* | *null* | *null* | *null* | 2 | 2 | 0.333 | research |
| 1 | 2 | *null* | *null* | *null* | *null* | 3 | 1 | 0.666 | research |
| 1 | 2 | *null* | *null* | *null* | *null* | 4 | 4 | 1.000 | research |

(d) Matching Table

**Fig. 5.** Temporary Matching Tables and the Matching Table.

Line 6 is to ensure that two matching leaf nodes have the same values. Lines 7–9 and 10–12 are used to filter out leaf nodes in the *old version* and *new version*, respectively, that already have been matched when the algorithm finds the fixed matching leaf nodes. Finally, the shifted matching leaf nodes are grouped by their *PID1* and *PID2* attributes.

At this point of time, we have two sets of matching leaf nodes groups, that is, one from fixed matching leaf nodes and another from shifted matching leaf nodes. The next step is to merge these sets of matching leaf nodes groups. Figure 4(d) depicts the SQL query to merge two sets of matching leaf nodes groups. We only need to use "`UNION`" operator (line 7) to merge these sets. The final step is to update the information of matching leaf nodes groups. We update the values of the *Total*, *Counter*, and *Score* attributes using the three SQL queries as depicted in Figures 4(e)–4(g). Suppose we have two set of leaf nodes, $\mathcal{G}_1$ and $\mathcal{G}_2$, whose parent nodes are $i_1$ and $i_2$, respectively, where $\mathcal{G}_1 \Leftrightarrow \mathcal{G}_2$. Then, the value of the *Total* attribute is equal to $(|t_1|+|t_2|)$, where $|t_1|$ and $|t_2|$ are the numbers of leaf nodes whose parent nodes are $i_1$ and $i_2$, respectively. That is, lines 3–6 and lines 7–10 in Figure 4(e) are used to calculate the values of $|t_1|$ and $|t_2|$, respectively. The value of the *Counter* attribute is equal to $(2|A|+|B|)$, where $|A|$ and $|B|$ are the numbers of fixed and shifted matching leaf nodes in $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively. Similarly, lines 3–6 and lines 8–12 in Figure 4(f) are used to calculate the

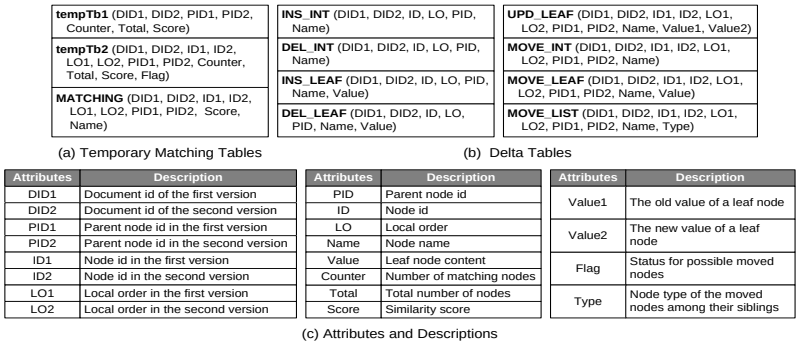



**Fig. 6.** Temporary and Delta Table Descriptions.

values of $2|A|$ and $|B|$, respectively. Finally, the value of the *Score* attribute is equal to $\frac{2|A|+|B|}{|t_1|+|t_2|}$ as defined in the preceding section.

The results of the *findMatchingLeafNodesGroups* algorithm are a temporary table $M_{c_x}$ in which the information of matching groups of non-inlined leaf nodes $c_x$ are stored. The schema of the $M_{c_x}$ table is the same as the one of the `tempTb1` table as depicted in Figure 6(a). The semantics of attributes of the `tempTb1` table are depicted in Figure 6(c). For instance, in our example, the "interest" node is a non-inlined leaf nodes. The algorithm will generate the $M_{interest}$ table as depicted in Figure 5(a).

### 3.2 Bottom-up Matching Phase

The next step is to propagate the matchings in bottom-up fashion (lines 5–14, Figure 3). First, the algorithm determines the highest level of the non-inlined internal nodes in DTD $U$ (line 5). Then, it starts to find best matching internal nodes in bottom-up fashion. There are two sub steps, that is, finding matching internal nodes (line 11) and determining best matching subtrees (line 12) by finding *best matching configurations*.

**Finding Matching Internal Nodes.** This phase is similar to the one discussed in [4]. Figure 7 depicts the SQL queries used to find matching internal nodes. Observe that these SQL queries are similar to the ones in [4]. The only difference is that in OXONE we include the *LocalOrder* attribute when we project the result of the SQL queries. The details on how to replace "`[moreConditions]`" (line 11, Figure 7(b)) can be found in [4]. The matching internal node $i_w$ is stored in a temporary matching table $M_{i_w}$, where $i_w \in \mathbb{N}_i$. The schema of the $M_{i_w}$ table is the same as the one of the `tempTb2` table as depicted in Figure 6(a). The semantics of attributes of the `tempTb2` table are depicted in Figure 6(c). For example, the matching "staff" node will be stored in the $M_{staff}$ table (Figure 5(b)).

**Finding Best Matching Internal Nodes.** The task in this step is to find *best matching configurations* that facilitate us to find best matching internal nodes. Recall that an internal node in $T_1$ can be matched to more than one internal nodes in $T_2$, and vice versa. The problem of finding best matching configuration is similar to the problem of finding *maximum weighted bipartite matching*. In our implementation, we use the Hungarian method [5] that addresses the problem of finding maximum weighted bipartite

```
1   INSERT INTO [tempTb]
2   SELECT
3     A1.DOC_ID AS DID1, A2.DOC_ID AS DID2,
4     A1.ID AS ID1, A2.ID AS ID2,
5     A1.LOCALORDER AS LO1, A2.LOCALORDER AS LO2,
6     A1.PID AS PID1, A2.PID AS PID2,
7     0 AS COUNTER, 0 AS TOTAL, 0 AS SCORE, 0 AS FLAG
8   FROM [tempMChild] AS A, [tbName] AS A1, [tbName] AS A2
9   WHERE
10    A.DID1 = doc_id1 AND A.DID2 = doc_id2 AND
11    A1.DOC_ID = doc_id1 AND A2.DOC_ID = doc_id2 AND
12    A1.ID = A.PID1 AND  A2.ID = A.PID2 AND
13    NOT EXISTS
14      (SELECT ID1, ID2 FROM [tempTb] AS B
15       WHERE B.DID1 = doc_id1 AND B.DID2 = doc_id2 AND
16             B.ID1 = A.ID1 AND B.ID2 = A.ID2)
17 GROUP BY A1.DOC_ID, A2.DOC_ID, A1.PID, A2.PID, A1.ID, A2.ID
```

(a) Finding Matching Internal Nodes (1)

```
1   INSERT INTO [tempTb]
2   SELECT
3     A1.DOC_ID AS DID1, A2.DOC_ID AS DID2,
4     A1.ID AS ID1, A2.ID AS ID2,
5     A1.LOCALORDER AS LO1, A2.LOCALORDER AS LO2,
6     A1.PID AS PID1, A1.PID AS PID2,
7     0 AS COUNTER, 0 AS TOTAL, 0 AS SCORE, 0 AS FLAG
8   FROM [tbName] AS A1, [tbName] AS A2
9   WHERE
10    A1.DOC_ID = doc_id1 AND A2.DOC_ID = doc_id2 AND
11    [moreConditions] AND
12    NOT EXISTS
13     (SELECT ID1, ID2 FROM [tempTb] AS B
14      WHERE B.DID1 = doc_id1 AND
15         B.DID2 = doc_id2 AND B.ID1 = A.ID1 AND B.ID2 = A.ID2)
16 GROUP BY A1.DOC_ID, A2.DOC_ID, A1.PID, A2.PID, A1.ID, A2.ID
```

(b) Finding Matching Internal Nodes (2)

**Fig. 7.** SQL Queries for Finding Matching Internal Nodes.

matching. The algorithm for finding best matching configurations is similar to the one discussed in [4] except for the following differences. After we determine the best matching configurations, the algorithm annotates the matching internal nodes whose parent nodes are not used in the best matching configuration by setting the *Flag* attribute in the $M_{i_w}$ table to "1". The annotations mean that these subtrees may be moved to different parent nodes. Note that in [4] such matching nodes are directly deleted. Observe that we also need to update the values of the *Counter*, *Total*, and *Score* attributes accordingly as initially their values are equal to "0".

### 3.3 Collecting Best Matching Internal Nodes Phase

The result of the previous step is the best matching internal nodes partitioned in several relations. The objectives of this step are to merge/collect the best matching internal nodes from different relations and to determine the best matching inlined internal nodes. Observe that the moved subtree candidates are also in the temporary matching tables. The values of the *Flag* attribute of moved subtree candidates in the temporary matching tables are equal to "1". The algorithm and SQL queries for collecting best matching internal nodes are similar to the ones presented in [4] except for the following difference. In OXONE, we need to filter out the moved node candidates from being considered as best matching internal nodes. They can be filtered out by adding a condition "`FLAG = 0`" in the SQL queries. In addition, we need to include the *LocalOrder* attribute when we project the result of the SQL queries. The best matching internal nodes are stored in the `MATCHING` table. The semantics of the `MATCHING` table is depicted in Figure 6. For example, given the $M_{univStaff}$ and $M_{staff}$ tables (Figures 5(b) and 5(c), respectively) and the relations containing the shredded XML documents, the `MATCHING` table is depicted in Figure 5(d). The `MATCHING` table keeps the best matching internal nodes of two XML documents that will be used to detect the changes (Phase 2).

| DID1 | DID2 | ID | LO | PID | Name |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 3 | 1 | staff |
| 1 | 2 | null | null | 3 | research |

(a) INS_INT Table

| DID1 | DID2 | ID | LO | PID | Name |
|---|---|---|---|---|---|
| 1 | 2 | 1 | 1 | 1 | staff |
| 1 | 2 | null | null | 1 | research |

(b) DEL_INT Table

| DID1 | DID2 | ID1 | ID2 | LO1 | LO2 | PID1 | PID2 | Name | Value1 | Value2 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | null | null | null | null | 3 | 1 | rank | Assoc Prof | Prof |

(e) UPD_LEAF Table

| DID1 | DID2 | ID | LO | PID | Name | Value | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 1 | interest | Information Retrieval | |
| 1 | 2 | null | null | 3 | name | Steve | # |
| 1 | 2 | null | null | 3 | rank | Asst Prof | # |
| 1 | 2 | 4 | 1 | 3 | interest | Semantic Web | |
| 1 | 2 | null | null | 1 | rank | Prof | # |

(c) INS_LEAF Table

| DID1 | DID2 | ID | LO | PID | Name | Value | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | null | null | 1 | name | Smith | # |
| 1 | 2 | null | null | 1 | rank | Assoc Prof | # |
| 1 | 2 | 1 | 1 | 1 | interest | Web Mining | |
| 1 | 2 | 2 | 2 | 1 | interest | Multimedia Mining | |
| 1 | 2 | 3 | 3 | 2 | interest | Data Mining | |
| 1 | 2 | null | null | 3 | rank | Assoc Prof | # |

(d) DEL_LEAF Table

**Fig. 8.** Delta Tables.

```
1  INSERT INTO UPD_LEAF
2  SELECT DISTINCT doc_id1 AS DID1, doc_id2 AS DID2,
3    NULL AS ID1, NULL AS ID2, NULL AS LO1, NULL AS LO2,
4    I1.ID AS PID1, I2.ID AS PID2, '[nodeName]' AS NAME,
5    I1.[attrName] AS VALUE1, I2.[attrName] AS VALUE2
6  FROM [parentTbName] AS I1, [parentTbName] AS I2
7  WHERE
8    I1.DOC_ID = doc_id1 AND I2.DOC_ID = doc_id2 AND
9    I1.[attrName] IS NOT NULL AND I2.[attrName] IS NOT NULL AND
10   I1.[attrName] != I2.[attrName] AND
11   EXISTS
12     (SELECT * FROM MATCHING AS B
13      WHERE DID1 = doc_id1 AND DID2 = doc_id2 AND
14        B.NAME = '[parentNodeName]' AND
15        B.ID1 = I1.ID AND B.ID2 = I2.ID )
```

(a) Update of Inlined Leaf Nodes

```
1  INSERT INTO UPD_LEAF
2  SELECT DISTINCT doc_id1 AS DID1, doc_id2 AS DID2,
3    D.ID AS ID1, I.ID AS ID2, D.LO AS LO1,
4    I.LO AS LO2, D.PID AS PID, I.PID AS PID2,
5    D.NAME, D.VALUE AS VALUE1, I.VALUE AS VALUE2
6  FROM INS_LEAF AS I, DEL_LEAF AS D, MATCHING AS M
7  WHERE
8    I.DID1 = doc_id1 AND I.DID2 = doc_id2 AND
9    D.DID1 = doc_id1 AND D.DID2 = doc_id2 AND
10   M.DID1 = doc_id1 AND M.DID2 = doc_id2 AND
11   M.ID1 = D.PID AND M.ID2 = I.PID AND
12   M.NAME = '[parentNodeName]' AND
13   I.NAME = '[nodeName]' AND
14   D.NAME = '[nodeName]' AND
15   I.VALUE != D.VALUE AND I.LO = D.LO
```

(b) Update of Non-inlined Leaf Nodes

**Fig. 9.** SQL Queries for Detecting Updated Leaf Nodes.

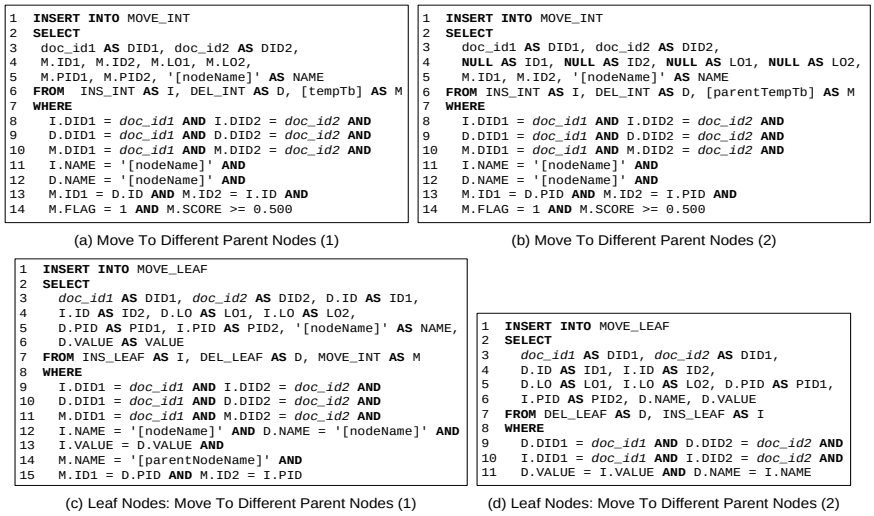
# 4 Change Detection Phase

In this section, we discuss how the changes are detected by OXONE after the best matching subtrees are determined. We detect the insertion, deletion, update, and move operations as highlighted in [1]. Note that we do not elaborate on the detection of inserted and deleted nodes (subtrees) here as the SQL queries are similar to the ones presented in [4]. The only difference is that in OXONE we include the "*LocalOrder*" attribute in the projection of the result. The detected inserted and deleted internal nodes are stored in the `INS_INT` and `DEL_INT` tables, respectively. Similarly, the detected inserted and deleted leaf nodes are stored in the `INS_LEAF` and `DEL_LEAF` relations, respectively. The semantics of these relations and corresponding examples (based on XML documents in Figure 1) are given in Figures 6 and 8, respectively. Note that the updated leaf nodes are also detected during the detection of inserted and deleted nodes as they can be decomposed into pairs of deleted and inserted leaf nodes. "`[param]`" in the SQL queries (Figures 9 and 10) used in the later discussion will be replaced the parameter *param* that is similar to the one defined in the *findBestMatchingSubtrees* algorithm.

## 4.1 Content Updates of Leaf Nodes

Intuitively, the updated leaf nodes are the leaf nodes that are available in both versions and have the same node names, but have different values, and their parent nodes are best matching internal nodes. In OXONE, the updated leaf nodes are detected after the inserted and deleted leaf nodes are detected. We classify the update operations of non-inlined leaf nodes into the *absolute update operations* and the *relative update operations*. 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. For inlined leaf nodes, we only have absolute update operations as they occur once under the same parent nodes.

**Inlined Leaf Nodes.** The SQL query in Figure 9(a) is used to determine the updated inlined leaf nodes. Lines 9–10 are used to ensure that the updated inlined leaf nodes are available in both versions (line 9) and they have different values (line 10). Lines 11–15 are used to guarantee that the parent nodes of the updated inlined leaf nodes are best matching internal nodes. The result of the SQL query depicted in Figure 9(a) is stored in



```
1  INSERT INTO MOVE_INT
2  SELECT
3   doc_id1 AS DID1, doc_id2 AS DID2,
4   M.ID1, M.ID2, M.LO1, M.LO2,
5   M.PID1, M.PID2, '[nodeName]' AS NAME
6  FROM  INS_INT AS I, DEL_INT AS D, [tempTb] AS M
7  WHERE
8    I.DID1 = doc_id1 AND I.DID2 = doc_id2 AND
9    D.DID1 = doc_id1 AND D.DID2 = doc_id2 AND
10   M.DID1 = doc_id1 AND M.DID2 = doc_id2 AND
11   I.NAME = '[nodeName]' AND
12   D.NAME = '[nodeName]' AND
13   M.ID1 = D.ID AND M.ID2 = I.ID AND
14   M.FLAG = 1 AND M.SCORE >= 0.500
```

(a) Move To Different Parent Nodes (1)

```
1  INSERT INTO MOVE_INT
2  SELECT
3    doc_id1 AS DID1, doc_id2 AS DID2,
4    NULL AS ID1, NULL AS ID2, NULL AS LO1, NULL AS LO2,
5    M.ID1, M.ID2, '[nodeName]' AS NAME
6  FROM INS_INT AS I, DEL_INT AS D, [parentTempTb] AS M
7  WHERE
8    I.DID1 = doc_id1 AND I.DID2 = doc_id2 AND
9    D.DID1 = doc_id1 AND D.DID2 = doc_id2 AND
10   M.DID1 = doc_id1 AND M.DID2 = doc_id2 AND
11   I.NAME = '[nodeName]' AND
12   D.NAME = '[nodeName]' AND
13   M.ID1 = D.PID AND M.ID2 = I.PID AND
14   M.FLAG = 1 AND M.SCORE >= 0.500
```

(b) Move To Different Parent Nodes (2)

```
1  INSERT INTO MOVE_LEAF
2  SELECT
3    doc_id1 AS DID1, doc_id2 AS DID2, D.ID AS ID1,
4    I.ID AS ID2, D.LO AS LO1, I.LO AS LO2,
5    D.PID AS PID1, I.PID AS PID2, '[nodeName]' AS NAME,
6    D.VALUE AS VALUE
7  FROM INS_LEAF AS I, DEL_LEAF AS D, MOVE_INT AS M
8  WHERE
9    I.DID1 = doc_id1 AND I.DID2 = doc_id2 AND
10   D.DID1 = doc_id1 AND D.DID2 = doc_id2 AND
11   M.DID1 = doc_id1 AND M.DID2 = doc_id2 AND
12   I.NAME = '[nodeName]' AND D.NAME = '[nodeName]' AND
13   I.VALUE = D.VALUE AND
14   M.NAME = '[parentNodeName]' AND
15   M.ID1 = D.PID AND M.ID2 = I.PID
```

(c) Leaf Nodes: Move To Different Parent Nodes (1)

```
1  INSERT INTO MOVE_LEAF
2  SELECT
3    doc_id1 AS DID1, doc_id2 AS DID1,
4    D.ID AS ID1, I.ID AS ID2,
5    D.LO AS LO1, I.LO AS LO2, D.PID AS PID1,
6    I.PID AS PID2, D.NAME, D.VALUE
7  FROM DEL_LEAF AS D, INS_LEAF AS I
8  WHERE
9    D.DID1 = doc_id1 AND D.DID2 = doc_id2 AND
10   I.DID1 = doc_id1 AND I.DID2 = doc_id2 AND
11   D.VALUE = I.VALUE AND D.NAME = I.NAME
```

(d) Leaf Nodes: Move To Different Parent Nodes (2)

**Fig. 10.** SQL Queries for Detecting Moved Nodes.

the UPD_LEAF table. Its schema and semantics are depicted in Figures 6. Next, we need to delete the corresponding tuples of the updated inlined leaf nodes in the DEL_LEAF and INS_LEAF relations. This is because we have detected updated leaf nodes that are previously detected as pairs of deleted and inserted leaf nodes.

**Non-Inlined Leaf Nodes.** To detect the absolute updated non-inlined leaf nodes, OXONE executes the SQL query depicted in Figure 9(b). Observe that we join three tables, namely, the DEL_LEAF, INS_LEAF, and MATCHING tables. Recall that an updated leaf node can be decomposed as a pair of deleted and inserted leaf nodes. Line 13 is used to guarantee that the parent nodes of the deleted and inserted leaf nodes are the best matching internal nodes. The absolute updated leaf nodes must have the same node name and the same local order, but different values (lines 13–15). The result of the SQL query depicted in Figure 9(b) is stored in the UPD_LEAF table. We also need to delete the corresponding tuples of the updated non-inlined leaf nodes in the DEL_LEAF and INS_LEAF relations.

Next, OXONE determines the *relative* updated non-inlined leaf nodes by executing the SQL query depicted in Figure 9(b) after slight modifications as follows. We replace "I.LO = D.LO" with "I.LO $\neq$ D.LO". Recall that the relative updated leaf nodes must have the same node name, but different values and local orders. Note that while detecting relative updated non-inlined leaf nodes, the query may return *incorrect* results in some situations as follows. First, there is more than one relative updated non-inlined leaf node under the same parent nodes. Second, there are deletion/insertion and update of non-inlined leaf nodes occurred under the same parent nodes. Therefore, we rectify the results using the approach as discussed in [4]. The result of the SQL query depicted in Figure 9(b) (after slight modification) is also stored in the UPD_LEAF table. In our example, the UPD_LEAF table is depicted in Figure 8(e). The highlighted tuples in the INS_LEAF (Figure 8(c)) and DEL_LEAF (Figure 8(d)) tables will be deleted as they are the corresponding tuples of the updated leaf nodes.

### 4.2 Move Operation

The move operations are classified into *move among siblings* and *move to different parent nodes*. The algorithm for detecting the movement of nodes among their siblings is similar to the one presented in [2]. Hence, here we focus on *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). The moved internal nodes (leaf nodes) are best matching internal nodes (matching leaf nodes) whose parent nodes are not best matching internal nodes.

The SQL queries in Figures 10(a) and 10(b) are used to find the moved *non-inlined* and *inlined* internal nodes that are moved to different parent nodes. Note that we only consider the moved internal nodes that have similarity scores equal or greater than "0.500". Note that this "threshold" can be defined by users based on application requirements. Otherwise, they are detected as pairs of deleted and inserted internal nodes. If an internal node $i$ is moved to different parents, then, intuitively, the subtree rooted at node $i$ is also moved. That is, we need to detect the moved leaf nodes that are the descendants of the moved internal nodes. Figure 10(c) is used to find the moved *non-inlined* leaf nodes that are the descendants of the moved internal nodes. To find the *inlined* ones, we used the modified SQL query of the SQL query depicted in Figure 10(c). We replace "`ID1`" and "`ID2`" in line 10 with "`PID1`" and "`PID2`" respectively. Note that we need to delete the corresponding tuples of the moved nodes that are stored in the `DEL_INT`, `INS_INT`, `DEL_LEAF`, and `INS_LEAF` tables. Observe that some leaf nodes can also be moved to be the child nodes of different parent nodes. These moved leaf nodes are not the descendants of the moved internal nodes. Figure 10(d) is used to find the moved leaf nodes that are not the descendants of the moved internal nodes. Note that we also need to delete the corresponding tuples of the moved leaf nodes that are stored in the `DEL_LEAF`, and `INS_LEAF` tables.

## 5 Performance Study

We have implemented OXONE entirely in Java. We use Microsoft SQL Server 2000 for storing XML documents before the changes are detected. The experiments were conducted on a Microsoft Windows XP Professional machine having Pentium 4 1.7 GHz processor with 512 MB of memory. We used a set of synthetic XML data based on SIGMOD DTD (http://www.sigmod.org/record/). The characteristics of the data sets are depicted in Figure 11. The second versions of the XML documents are generated by using our XML change generator. We compared the performance of OXONE to the Java version of X-Diff [8] (downloaded from http://www.cs.wisc.edu/∼yuanwang/xdiff.html), schema-oblivious relational-based approach for ordered XML change detection in [2] (called XANDY–O), and C version of XyDiff [1] (downloaded from http://pauillac.inria.fr/cdrom/www/xydiff/index-eng.htm). Note that despite our best efforts (including contacting the authors), we could not get the Java version of XyDiff. The C version of XyDiff was run in a Pentium 4 1.7 GHz processor with 512 MB of

| Dataset Code | Number of Nodes | | | Filesize (KB) |
|---|---|---|---|---|
| | Internal | Leaf | Total | |
| SIGMOD-01 | 73 | 258 | 331 | 13 |
| SIGMOD-02 | 117 | 427 | 554 | 21 |
| SIGMOD-03 | 187 | 703 | 890 | 34 |
| SIGMOD-04 | 389 | 1,437 | 1,826 | 70 |
| SIGMOD-05 | 567 | 2,151 | 2,718 | 104 |
| SIGMOD-06 | 983 | 3,734 | 4,717 | 180 |
| SIGMOD-07 | 1,801 | 6,993 | 8,794 | 337 |
| SIGMOD-08 | 3,883 | 14,983 | 18,866 | 721 |

| Dataset Code | Number of Nodes | | | Filesize (KB) |
|---|---|---|---|---|
| | Internal | Leaf | Total | |
| SIGMOD-09 | 7723 | 30,002 | 37,725 | 1,444 |
| SIGMOD-10 | 18,067 | 71,256 | 89,323 | 3,431 |
| SIGMOD-11 | 34,845 | 137,909 | 172,754 | 6,635 |
| SIGMOD-12 | 58,587 | 231,952 | 290,539 | 11,167 |
| SIGMOD-13 | 71,991 | 283,930 | 355,921 | 13,688 |
| SIGMOD-14 | 91,604 | 361,085 | 452,689 | 17,398 |
| SIGMOD-15 | 125,411 | 494,812 | 620,223 | 23,816 |

**Fig. 11.** Data Sets.

memory with Red Hat Linux 9 operating system. Note that as the Java version is in general slower than the C version, the execution times of XyDiff will differ by a constant factor in comparison with X-Diff.

***Execution Time vs Number of Nodes***. In this set of experiments, we analyze the performance of OXONE for different number of nodes. The percentages of change is set to "9%". Figure 12(a) depicts the performance of Phase 1 in our approaches. Observe that the performances of OXONE and XANDY–O are comparable up to data set `SIGMOD-05`. For larger data set, OXONE outperforms XANDY–O (up to 20.5 times). Note that the performance of XANDY–O is adversely affected with increase in number of nodes and for datasets larger than `SIGMOD-12`, XANDY–O fail to return results in 100,000 seconds. Hence, we do not plot the result of XANDY–O for data sets larger than `SIGMOD-12`. The performance of Phase 2 is depicted in Figure 12(b). In this case, OXONE is faster than XANDY–O (up to 81.88 times).

Figure 12(c) depicts the overall performance of our approaches. XyDiff is up to 3.5 times faster than OXONE for the first three data sets. After that the performance of XyDiff is comparable to the one of OXONE. However, our approach is more scalable as XyDiff fails to detect the changes to data sets larger than `SIGMOD-12` as its process was killed by the Linux kernel. In addition, 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. X-Diff, on the other hand, is only able to detect the changes up to `SIGMOD-06` due to lack of main memory. X-Diff outperforms OXONE for the first three data sets (up to 8.15 times). For larger data sets, OXONE is up to 43.7 times faster than X-Diff. Note that the performances of XANDY–O and OXONE is slower than main memory-based approaches for smaller data sets as the database I/O cost is more expensive. Also, overall OXONE is up to 22 times faster than XANDY–O.

***Result Quality***. Next, we examine the result quality of OXONE, XANDY–O, and XyDiff. The result quality is defined as the ratio between the number of edit operations in the deltas detected by an approach and the one in the *optimal* deltas. Note that an optimal delta consists of minimum number of edit operations [8]. Also, we do not show the result quality of X-Diff as it is not designed for ordered change detection. We use a small data set with 100 nodes and generate the second version with various percentages of changes (2%–12%). Figure 12(d) depicts the result quality comparison results. Observe that the result quality of OXONE and of XANDY–O are comparable. Also, the result qualities of OXONE and XANDY–O are significantly better than that of XyDiff. In XyDiff's deltas, there are some unnecessary move operations, and, in some case, XyDiff mismatches the best matching subtrees. For instance, consider the example depicted in Figure 13. The delta detected by OXONE contains $delete$(1) and $update$(10, "Asst

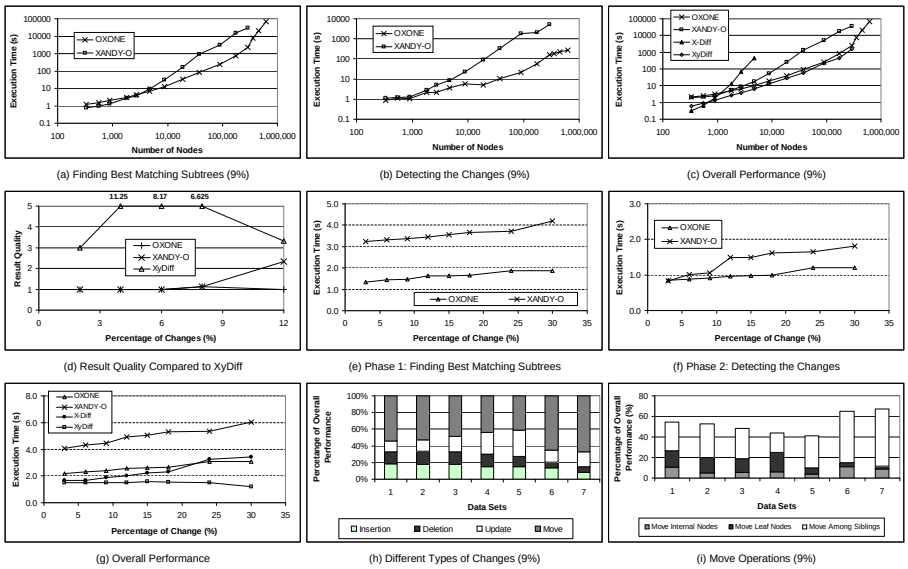



**Fig. 12.** Experimental Results.

Prof", "Assoc Prof"). However, the delta generated by XyDiff contains $move(9, 1, 2)$ ("move node 9 to the second child node of node 1"), $delete(8)$, and $delete(2)$ which is semantically incorrect.

***Execution Time vs Percentages of Changes***. In this section, we shall observe the effects of percentage of changes to the performances of XANDY–O, OXONE, and X-Diff. We use "`Sigmod-03`" data set. Observe that the percentages of changes are equally distributed to different types of changes. Figure 12(e) depicts the performance of Phase 1 of XANDY–O and OXONE for different percentage of changes. The performances of XANDY–O and OXONE are affected by percentages of changes. When the percentage of changes is increased by 1%, the performances of XANDY–O and OXONE become, on average, 0.95% and 1.43% slower, respectively. The performances of Phase 2 of XANDY–O and OXONE for different percentages of changes are depicted in Figure 12(f). OXONE is up to 1.62 times faster than XANDY–O. Figure 12(g) shows the overall performance of XANDY–O and OXONE for different percentages of changes. We notice that XyDiff is up to 2.59 times faster than OXONE.

***Different Types of Changes***. In this set of experiments, we study the affect of different types of changes on the running time. We used first seven data sets and set the

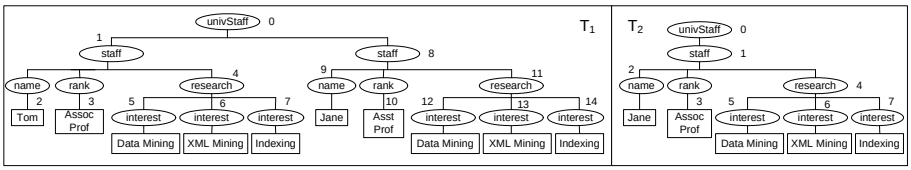



**Fig. 13.** Result Quality: Example.

percentage of changes to 9%. Figure 12(h) depicts the proportion of execution times of detecting insertion, deletion, update, and move operations. Observe that detecting move operation takes up to 67.17% of the execution time of Phase 2. Figure 12(i) depicts the affect of different types of move operations on the running time. The execution time of detecting moves among siblings is significant compared to the one for detecting moved internal nodes and moved leaf nodes. It takes up to 55.68% of the execution time of Phase 2. Let us elaborate on this further. In detecting move among siblings, we compare the local order of each node. However, the local order can be changed due to the insertions/deletions of sibling nodes. Hence, we need to ensure that such local order changes are not considered in order to detect move among siblings correctly. The *adjustLocalOrder* function that is similar to the one in [2] is used to simulate the insertions/deletions of sibling nodes. Observe that an insertion/deletion of a sibling node can change more than one local orders of its sibling nodes. Hence, when there is an insertion/deletion of a sibling node, we need to adjust more than one local orders of its sibling nodes. That is, the cost of the *adjustLocalOrder* function is increased as the number of insertions/deletions is increased.

## 6 Conclusions and Future Work

In this paper, we present a relational-based approach (called OXONE) for detecting the changes on ordered XML documents using a schema-conscious approach. This work is motivated by the following observations. First, existing main memory-based ordered XML change detection techniques (XyDiff) produce poorer quality deltas compared to its unordered counterpart (X-Diff). Second, although existing relational-based ordered change detection approach such as XANDY–O can produce superior quality results, its performance is much slower than XyDiff and degrades significantly with increase in number of nodes. To the best of our knowledge, OXONE is the first approach that address these two limitations. Our experimental results show that OXONE is more scalable than existing state-of-the-art approaches. It has comparable performance with XyDiff and yet produce superior result quality. As parts of our future work, we would like to extend our framework so that it can handle recursive DTDs.

## References


1. G. COBENA, S. ABITEBOUL, A. MARIAN. Detecting Changes in XML Documents. *In ICDE*, 2002.
2. E. LEONARDI, S. S. BHOWMICK. XANDY: A Scalable Change Detection Technique for Ordered XML Documents Using Relational Databases. To appear in DKE Journal.
3. E. LEONARDI, S. S. BHOWMICK, S. MADRIA. XANDY: Detecting Changes on Large Unordered XML Documents Using Relational Databases. *In DASFAA*, China, 2005.
4. E. LEONARDI, S. S. BHOWMICK. Detecting Changes on Unordered XML Documents Using Relational Databases: A Schema-Conscious Approach. *In CIKM*, 2005.
5. C. PAPADIMITRIOU, K. STEIGLITZ. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, NJ, 1982.
6. J. SHANMUGASUNDARAM, K. TUFTE, C. ZHANG, G. HE, D. J. DEWITT, AND J. F. NAUGHTON Relational Databases for Querying XML Documents: Limitations and Opportunities. *The VLDB Journal*, 1999.
7. H. LU, H. JIANG, J. X. XU, G. YU ET AL. What Makes the Differences: Benchmarking XML Database Implementations. *In ACM TOIT*, 5(1), 2005.
8. Y. WANG, D. J. DEWITT, J. CAI. X-Diff: An Effective Change Detection Algorithm for XML Documents. *In ICDE*, Bangalore, 2003.