Detecting Changes on Unordered XML Documents Using Relational Databases: A Schema-Conscious Approach

Erwin Leonardi
pk909134@ntu.edu.sg

Sourav S. Bhowmick
assourav@ntu.edu.sg

School of Computer Engineering, Nanyang Technological University, Singapore

ABSTRACT

Recently, a number of main memory algorithms for detecting the changes to XML data has been proposed. These techniques suffer from scalability problem and fail to detect changes to large XML documents. As a result, several *relational approaches* have been proposed to detect the changes to XML documents by using relational databases. These approaches store the XML documents in the relational database and issue SQL queries (whenever appropriate) to detect the changes. All of these relational-based approaches use the *schema-oblivious* XML storage strategy for detecting the changes. However, there is growing evidence that schema-conscious storage approaches perform significantly better than schema-oblivious approaches as far as XML query processing is concerned. In this paper, we study a relational-based unordered XML change detection technique (called HELIOS) that uses a *schema-conscious* approach (Shared-Inlining) as the underlying storage strategy. HELIOS is up to 52 times faster than X-Diff [10] for large datasets (more than 1000 nodes). It is also up to 6.7 times faster than XANDY [6]. The result quality of deltas detected by HELIOS is comparable to the result quality of deltas detected by XANDY.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems – *Relational databases.*

General Terms: Algorithms, Design, Experimentation.

Keywords: XML, change management, change detection.

1. INTRODUCTION

Detecting changes to XML data is an important research problem. The XML change detection problem is related to the problem of detecting the changes 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 to *unordered* trees. Recently, a number of techniques for detecting the changes to XML data has been proposed. XyDiff [4] is a main-memory algorithm 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 [10] 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, 6], we have addressed this scalability problem by using the relational database system. In this approach, given the old and new versions of an XML document, we store both documents in relational database. Next, we issue a set of SQL queries to detect the changes. The relational approach-based XML change detection has potential to gain popularity due to its stability, scalability, and its wide spread usage in the commercial world. However, efficient change detection in this approach largely determined by the underlying storage approach. Particularly, there are two major approaches for storing XML documents in a relational database. In *schema-conscious approach*, a relational schema is created based on the DTD/schema of the XML documents. First, the cardinality of the relationships between the nodes of the XML document is established. Based on this information a relational schema is created. The structural information of XML data is modeled by using primary-key foreign-key joins in relational databases to model the parent-child relationships in the XML tree. 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. Also, number of tables is fixed in the relational schema and does not depend on the structural heterogeneity of XML documents.

```
<!ELEMENT catalog (item*)>
<!ELEMENT item (name,
   category?, price, details)>
<!ELEMENT details (info*)>
<!ELEMENT pname (#PCDATA)>
<!ELEMENT category (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT info (#PCDATA)>
```

(a) An example of DTD

(b) Relational schema
Document (Doc_ID, Doc_Name)
Catalog (*Doc_ID*, ID)
Item (*Doc_ID*, ID, *PID*, PName, Category, Price, Details)
Info (*Doc_ID*, ID, *PID*, Val)

Figure 1: XML Trees and XML in RDBMS.

Our previous approaches for detecting changes to XML data [3, 6] were all based on the schema-oblivious approaches. This is motivated by the fact that schema-oblivious approaches have the following two advantages. First, ability to handle XML schema changes better as there is no need to change the relational schema. Second, there is no need to modify SQL queries to detect changes even if the structure of XML data changes.

CIKM'05, October 31–November 5, 2005, Bremen, Germany.

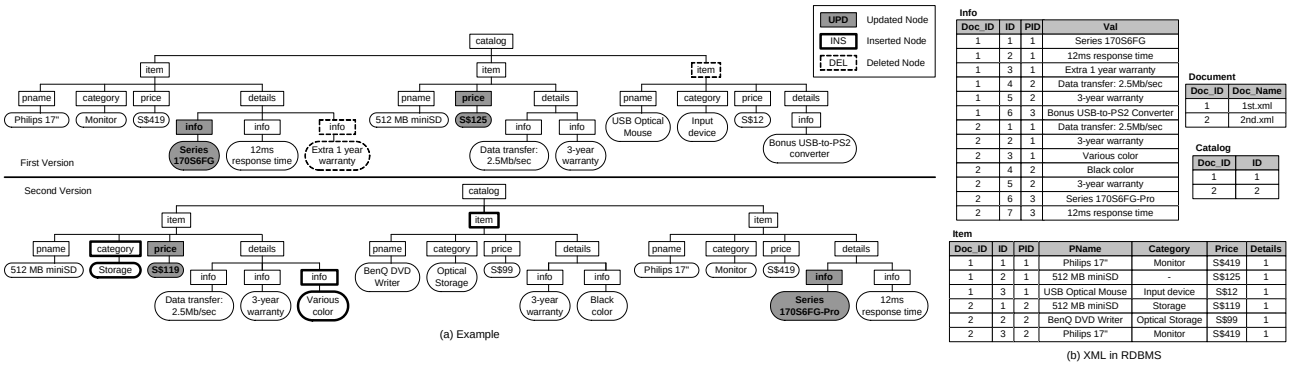


Figure 2: XML Trees and XML in RDBMS.

In this paper, we present a novel relational approach for detecting the changes to *unordered* XML documents called HELIOS (*sc***HE***ma-conscious xml-enab***L***ed change detect***IO***n* ***S****ystem*) using a *schema-conscious approach* (Shared-Inlining [8] in our case). Our effort is motivated by the fact that a growing body of work suggests that schema-conscious approaches perform better than schema-oblivious approaches as far as XML query processing is concerned [5, 9]. Hence, is it possible to design a schema-conscious XML change detection system that can accurately detect all types of changes yet outperform existing XML change detection approaches? In this paper, we address this issue. Note that the characteristics of schema-conscious approach raise certain challenges. For instance, in this approach no special relational schema needs to be designed as it can be generated on the fly based on the DTD of the XML document(s). That is, 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. In other words, given a specific schema-oblivious approach, our framework should be able to detect all changes accurately independent of the changes to the underlying relational schema due to different structure of XML documents. Note that the framework discussed in this paper is only for XML documents whose schemas do not contain recursive elements.

In our approach, first, we store two versions of XML documents in RDBMS by using Shared-Inlining schema [8] generated based on their DTD. Next, our approach starts to detect the changes in bottom-up fashion. Our approach consists of two phases, namely, the *finding the best matching subtrees* phase (Phase 1) and the *detecting the changes* phase (Phase 2). For each phase, the algorithm executes several SQL queries. Note that "[param]" in the SQL queries used in the later discussion will replaced the parameter *param* defined in the algorithm. The detected delta will be stored in several relations in RDBMS.

We have implemented the prototype of HELIOS using Java on top of Shared-Inlining [8], a schema-conscious storage strategy for XML documents. We compared HELIOS to XANDY [6], a published schema-oblivious unordered XML change detection system, and X-Diff [10], a published main memory-based approach. We observe that the overall performance of HELIOS performs up to 6.7 times faster than XANDY. Also, although X-Diff outperforms HELIOS for small data sets (less than 1000 nodes), for larger data sets, HELIOS is up to 52 times faster than X-Diff. X-Diff is unable to detect the changes on XML documents that have more than 5000 nodes due to lack of main memory. The *result quality* of HELIOS is comparable to the one of XANDY.

```
Input
  U : DTD of the XML documents
  Two versions of an XML document
      stored in RDBMS
Output
  the Matching table
   /* --- STEP 1 --- */
1  for all a_x in U do
2   tbName ← r_ax; tempTb ← M_ax;
3   findMatchingLeafNodes(tbName,
                          tempTb);
4  end for
   /* --- STEP 2 --- */
5  maxLevel = maximum level at which
       there is d_x in U
   /* bottom-up matching */
6  for lev = maxLevel down to 1 do
7    for all d_x at level lev do
8      childNode ← child(d_x);
9      tempMChild ← M_childNode;
10     tbName ← r_dx; tempTb ← M_dx;
       /* --- STEP 2.1 --- */
11     findMatchingInternalNodes(tbName,
                    tempTb, tempMChild);
       /* --- STEP 2.2 --- */
12     maximizeScore(d_x);
13   end for
14 end for
   /* --- STEP 3 --- */
15 root is the root node of U
16 Queue Q ←{root}
17 while (Q is not empty) do
18   q = Q.get();
19   Q ← the child internal
            nodes of q in U;
20   nodeName ← name(q); tempTb ← r_q;
21   parentNode ← parent(q);
22   parentNodeName ← name(parentNode);
23   attrName ← attribute(q);
24   retrieveMatching(nodeName, tempTb,
           parentNodeName, attrName);
25 end do
```

Figure 3: The `bestMatchingFinder` Algorithm.

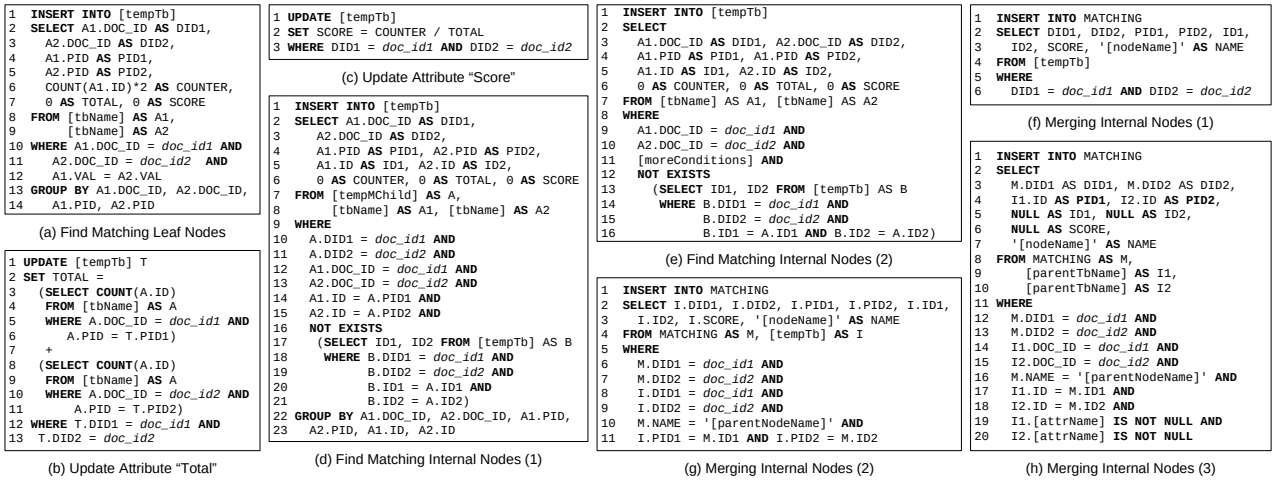
2. FINDING BEST MATCHING SUBTREES

The *Finding Best Matching Subtrees* phase is important because it facilitates the system in finding *minimum delta*. In this paper, the *minimum delta* is defined as a delta that has the least number of changed nodes. Given a DTD as depicted in Figure 1(a), we generate a Shared-Inlining schema. We extend the schema such that an internal node that has in-degree one and out-degree one in DTD tree is stored as Boolean attribute. This will enable us to detect the inserted/deleted of this type of internal nodes. For example, node "*details*" in the DTD. The modified schema is depicted in Figure 1(b). Note that we use the XML documents depicted in Figure 2(a) as our running example.

2.1 Preliminaries

We first present notations that will be used in the later discussion. Suppose we have a Shared-Inlining schema S generated based on DTD U. Schema S consists of a set of tables $R(S)$= $\{r_{n_1}, r_{n_2}, ..., r_{n_x}\}$, where n_p is the name of the relation. The nodes in DTD U are categorized into two types as follows.

- ***Inlined Nodes***. The *inlined node* is one that is stored as an attribute of the relation of its parent node. There are two types of inlined nodes, namely, *inlined leaf nodes* (denoted by a) and *inlined internal nodes* (denoted by b). The attribute that stores the information on an *inlined* node n is denoted as $attribute(n)$. For example, a leaf node "*price*" is also an inlined node as it is stored as attribute *Price* in the r_{item} relation.
- ***Non-inlined Nodes***. The *non-inlined* node is one that is stored as a relation. Similarly, there are two types of non-inlined nodes, namely, *non-inlined leaf nodes* (denoted by c) and *non-inlined internal nodes* (denoted by d). The relation that stores the information on a *non-inlined node* n is denoted as r_n. For



(a) Find Matching Leaf Nodes

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

(b) Update Attribute "Total"

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

(c) Update Attribute "Score"

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

(d) Find Matching Internal Nodes (1)

```
1  INSERT INTO [tempTb]
2  SELECT
3    A1.DOC_ID AS DID1, A2.DOC_ID AS DID2,
4    A1.PID AS PID1, A1.PID AS PID2,
5    A1.ID AS ID1, A2.ID AS ID2,
6    0 AS COUNTER, 0 AS TOTAL, 0 AS SCORE
7  FROM [tbName] AS A1, [tbName] AS A2
8  WHERE
9    A1.DOC_ID = doc_id1 AND
10   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
16            B.ID1 = A.ID1 AND B.ID2 = A.ID2)
```

(e) Find Matching Internal Nodes (2)

```
1  INSERT INTO MATCHING
2  SELECT I.DID1, I.DID2, I.PID1, I.PID2, I.ID1,
3    I.ID2, I.SCORE, '[nodeName]' AS NAME
4  FROM MATCHING AS M, [tempTb] AS I
5  WHERE
6    M.DID1 = doc_id1 AND
7    M.DID2 = doc_id2 AND
8    I.DID1 = doc_id1 AND
9    I.DID2 = doc_id2 AND
10   M.NAME = '[parentNodeName]' AND
11   I.PID1 = M.ID1 AND I.PID2 = M.ID2
```

(g) Merging Internal Nodes (2)

```
1  INSERT INTO MATCHING
2  SELECT DID1, DID2, PID1, PID2, ID1,
3    ID2, SCORE, '[nodeName]' AS NAME
4  FROM [tempTb]
5  WHERE
6    DID1 = doc_id1 AND DID2 = doc_id2
```

(f) Merging Internal Nodes (1)

```
1  INSERT INTO MATCHING
2  SELECT
3    M.DID1 AS DID1, M.DID2 AS DID2,
4    I1.ID AS PID1, I2.ID AS PID2,
5    NULL AS ID1, NULL AS ID2,
6    NULL AS SCORE,
7    '[nodeName]' AS NAME
8  FROM MATCHING AS M,
9      [parentTbName] AS I1,
10     [parentTbName] AS I2
11 WHERE
12   M.DID1 = doc_id1 AND
13   M.DID2 = doc_id2 AND
14   I1.DOC_ID = doc_id1 AND
15   I2.DOC_ID = doc_id2 AND
16   M.NAME = '[parentNodeName]' AND
17   I1.ID = M.ID1 AND
18   I2.ID = M.ID2 AND
19   I1.[attrName] IS NOT NULL AND
20   I2.[attrName] IS NOT NULL
```

(h) Merging Internal Nodes (3)

Figure 4: SQL Queries (1).

example, an internal node "*item*" is a non-inlined node as it is stored as r_{item} relation.

We now define some other symbols to facilitate exposition. Let L_v be a set of leaf nodes in the version v of an XML document. Let ℓ_v be a leaf node in the version v of an XML document, where $\ell_v \in L_v$. The textual content of ℓ_v is denoted by $value(\ell_v)$. A set of internal nodes in the version v of an XML document is denoted as I_v, and i_v denotes an internal node in version v of an XML document, where $i_v \in I_v$. The name and level of node n are denoted by $name(n)$ and $level(n)$ respectively. The parent node, child node, and ancestor node of n are denoted as $parent(n)$, $child(n)$, and $ancestor(n)$ respectively.

The `bestMatchingFinder` algorithm that is used to determine the best matching subtrees from two versions of an XML document consists of three steps: *finding matching leaf nodes*, *finding best matching internal nodes*, and *collecting best matching internal nodes*. We shall elaborate these steps in the later discussion.

2.2 Finding the Matching Leaf Nodes

The objective of this step is to determine the *matching leaf nodes*. Informally, the matching leaf nodes are ones that have the same node name, node level, and value. Formally, the matching leaf nodes are defined as follows. Let ℓ_{1_x} and ℓ_{2_y} be two leaf nodes in the first and second versions of an XML tree respectively. Then, ℓ_{1_x} and ℓ_{2_y} are **matching leaf nodes** (denoted as $\ell_{1_x} \leftrightarrow \ell_{2_y}$) if $name(\ell_{1_x}) = name(\ell_{2_y})$, $level(\ell_{1_x}) = level(\ell_{2_y})$, and $value(\ell_{1_x}) = value(\ell_{2_y})$. Note that we only match the *non-inlined* leaf nodes in this step. The *inlined* leaf nodes will be matched when we match internal nodes (Step 2). Suppose we have two versions of an XML document stored in RDBMS as depicted in Figure 2(b). We only have one *non-inlined* leaf node, namely, *info* node stored in the `Info` table. Let $n_{(p,q)}$ be a leaf node n stored in the r_n table with *doc_id*=p and *id*=q. We notice that there are four matching leaf nodes, namely, $info_{(1,2)} \leftrightarrow info_{(2,7)}$, $info_{(1,4)} \leftrightarrow info_{(2,1)}$, $info_{(1,5)} \leftrightarrow info_{(2,2)}$, and $info_{(1,5)} \leftrightarrow info_{(2,5)}$.

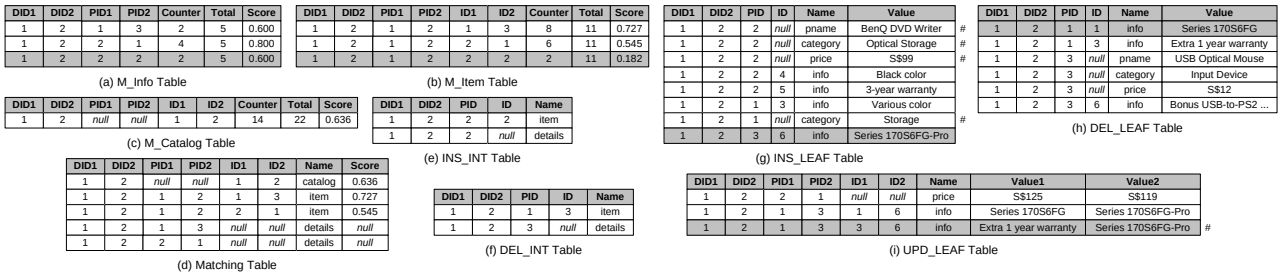
Instead of storing the matching leaf nodes directly, we group them according to their parent nodes and store these *matching groups* in a temporary table in order to reduce storage space requirement. Let $G_{(1,pid_1)} = \{\ell_{1_1}, \ell_{1_2}, ..., \ell_{1_x}\}$ and $G_{(2,pid_2)} = \{\ell_{1_1}, \ell_{2_2}, ..., \ell_{2_y}\}$ be two sets of *non-inlined* leaf nodes in the first and second versions of an XML document respectively, where $\forall \ell_{1_p} \in G_1$ have the parent node id pid_1, and $\forall \ell_{2_p} \in G_2$ have the parent node id pid_2. Then $G_{(1,pid_1)}$ and $G_{(2,pid_2)}$ are **matching groups** (denoted by $G_{(1,pid_1)} \Leftrightarrow G_{(2,pid_2)}$) if $\exists \ell_{1_p}\ \exists \ell_{2_q}$ such that $\ell_{1_p} \leftrightarrow \ell_{2_q}$ where $\ell_{1_p} \in G_{(1,pid_1)}$ and $\ell_{2_q} \in G_{(2,pid_2)}$. Recall the example depicted in Figure 2(b). Let $G_{(v,w)}$ be a matching group in which there is a set of leaf nodes with parent node id w in the version v. We have three matching groups, namely, $G_{(1,1)} \Leftrightarrow G_{(2,3)}$, $G_{(1,2)} \Leftrightarrow G_{(2,1)}$, and $G_{(1,2)} \Leftrightarrow G_{(2,2)}$.

After grouping the matching leaf nodes, we need to measure how similar the matching groups are by calculating their *similarity scores*. The *similarity score* $\Re$ of two matching groups G_1 and G_2 is defined as follows: $\Re(G_1, G_2) = \frac{2|G_1 \cap G_2|}{|G_1|+|G_2|}$, where $|G_1|+|G_2|$ is the total number of leaf nodes in G_1 and G_2, and $|G_1 \cap G_2|$ is the number of matching leaf nodes in G_1 and G_2. The value of similarity score is between 0 and 1. If two matching groups have no matching leaf nodes, then their similarity score is 0. The similarity score of two matching groups is equal to 1 if they are identical. For example, the similarity score of $G_{(1,1)}$ and $G_{(2,3)}$ is equal to "0.600" as $|G_{(1,1)} \cap G_{(2,3)}| = 1$ and $|G_{(1,1)}| + |G_{(2,3)}| = 5$.

Given two versions of an XML document shredded in RDBMS, we use three SQL queries encapsulated in the `findMatchingLeafNodes` algorithm to find matching groups . The first SQL query is depicted in Figure 4(a). This query is used to find the matching leaf nodes (line 12) and group them into matching groups (lines 13-14). This SQL query is also used to calculate the number of matching leaf nodes in the matching groups (line 6). Figure 4(b) depicts the SQL query that is used to calculate total number of leaf nodes in the matching groups. The third SQL query that is used to calculate the similarity scores of the matching groups is depicted in Figure 4(c). Given the `Info` table depicted in Figure 2(b), the `findMatchingLeafNodes` algorithm results the `M_Info` table that stores the matching groups. The `M_Info` table is the instance of the `tempTb1` table as depicted in Figures 6(a) and (b).

2.3 Finding Best Matching Internal Nodes

The objective of this step is to determine *best matching non-inlined internal nodes* between two XML documents. Lines 5-14 in Figure 3 are used to find *best matching internal nodes*. The algorithm works as follows. First, the algorithm determines the highest level of the internal nodes in DTD U. Then, the algorithm starts to find best matching internal nodes in bottom-up fashion. There are two sub steps as follows. First, finding matching internal nodes (line 11, Figure 3). Second, determining best matching subtrees (line 12, Figure 3).



(a) M_Info Table

DID1	DID2	PID1	PID2	Counter	Total	Score
1	2	1	3	2	5	0.600
1	2	2	1	4	5	0.800
1	2	2	2	2	5	0.600

(b) M_Item Table

DID1	DID2	PID1	PID2	ID1	ID2	Counter	Total	Score
1	2	1	2	1	3	8	11	0.727
1	2	1	2	2	1	6	11	0.545
1	2	1	2	2	2	2	11	0.182

(c) M_Catalog Table

DID1	DID2	PID1	PID2	ID1	ID2	Counter	Total	Score
1	2	*null*	*null*	1	2	14	22	0.636

(d) Matching Table

DID1	DID2	PID1	PID2	ID1	ID2	Name	Score
1	2	*null*	*null*	1	2	catalog	0.636
1	2	1	2	1	3	item	0.727
1	2	1	2	2	1	item	0.545
1	2	1	3	*null*	*null*	details	*null*
1	2	2	1	*null*	*null*	details	*null*

(e) INS_INT Table

DID1	DID2	PID	ID	Name
1	2	2	2	item
1	2	2	*null*	details

(f) DEL_INT Table

DID1	DID2	PID	ID	Name
1	2	1	3	item
1	2	3	*null*	details

(g) INS_LEAF Table

DID1	DID2	PID	ID	Name	Value	
1	2	2	*null*	pname	BenQ DVD Writer	#
1	2	2	*null*	category	Optical Storage	#
1	2	2	*null*	price	S$99	#
1	2	2	4	info	Black color	
1	2	2	5	info	3-year warranty	
1	2	1	3	info	Various color	
1	2	1	*null*	category	Storage	#
1	2	3	6	info	Series 170S6FG-Pro	

(h) DEL_LEAF Table

DID1	DID2	PID	ID	Name	Value	
1	2	1	1	info	Series 170S6FG	
1	2	1	3	info	Extra 1 year warranty	
1	2	3	*null*	pname	USB Optical Mouse	#
1	2	3	*null*	category	Input Device	#
1	2	3	*null*	price	S$12	#
1	2	3	6	info	Bonus USB-to-PS2 ...	

(i) UPD_LEAF Table

DID1	DID2	PID1	PID2	ID1	ID2	Name	Value1	Value2	
1	2	2	1	*null*	*null*	price	S$125	S$119	
1	2	1	3	1	6	info	Series 170S6FG	Series 170S6FG-Pro	
1	2	1	3	3	6	info	Extra 1 year warranty	Series 170S6FG-Pro	#

Figure 5: Temporary Matching, Matching, and Delta Tables.

Finding matching internal nodes. Informally, two internal nodes are *matching internal nodes* if they have at least one matching leaf nodes that are their descendants. Let i_1 and i_2 be two internal nodes from the old and new versions of an XML document respectively. Then, i_1 and i_2 are **matching internal nodes** (denoted by $i_1 \doteq i_2$) if the following conditions are satisfied: 1) $name(i_1) = name(i_2)$, 2) $level(i_1) = level(i_2)$, and 3) $\exists \ell_{1_x} \exists \ell_{2_y}$ such that $\ell_{1_x} \leftrightarrow \ell_{2_y}$, where $i_1 = ancestor(\ell_{1_x})$ and $i_2 = ancestor(\ell_{2_y})$.

For each *non-inlined* internal node d_x, the `bestMatching-Finder` algorithm invokes the *findMatchingInternalNodes* algorithm. The *findMatchingInternalNodes* algorithm executes the SQL query as depicted in Figure 4(d). This SQL query is used to find matching internal nodes from the temporary matching relations of the child nodes of the input node. Lines 14-15 are used to ensure that the parent-child relationship between the matching nodes and the internal nodes that are going to be matched. The result will be grouped by their parent nodes (lines 22-23). Lines 16-21 are used to avoid duplicate matching internal nodes. The result of the SQL query in Figure 4(d) is stored in a temporary matching table M_{d_x} that is the instance of the `tempTb2` table as depicted in Figures 6(a) and (b).

If node d_x has *inlined* child leaf nodes, then the *findMatchingInternalNodes* algorithm also executes the SQL query depicted in Figure 4(e). Before executing the SQL query, the algorithm prepares one more parameter, namely, *moreConditions* that is used to replace a part of the SQL query. Formally, the value of *moreConditions* is as follows. Suppose we have two internal nodes i_1 and i_2 from the old and new versions of XML documents respectively. Nodes i_1 and i_2 are matching internal nodes ($i_1 \doteq i_2$) if $\exists a_{1_z} \exists a_{2_z}$ such that $value(a_{1_z}) = value(a_{2_z})$, where $a_{1_z} = child(i_1)$, $a_{2_z} = child(i_2)$, and $attribute(a_{1_z}) = attribute(a_{2_z})$. For example, given the "`Item`" table, the value of parameter *moreConditions* is equal to "`(A1.PName = A2.PName OR A1.Category = A2.Category OR A1.Price = A2.Price)`". The result of the SQL query in Figure 4(e) is also stored in a temporary matching table M_{d_x}.

Determining the best matching internal nodes. The *best matching internal nodes* are formally defined as follows. Let $i \in I_1$ be an internal node in the old version of an XML document. Let $Z \subseteq I_2$ be a set of internal nodes in new version, where $\forall z_p \in Z$ such that $i \doteq z_p$. i and z_q are **best matching internal nodes** (denoted by $i \doteqdot z_q$) iff $(\Re(i, z_q) > \Re(i, z_p)) \ \forall \ 0 < p \leq |Z|$ and $q \neq p$. For example, in the `M_Item` table we notice that $item_{(1,2)}$ can be matched to $item_{(2,1)}$ and $item_{(2,2)}$. The similarity score of matching *catalog* nodes ($catalog_{(1,1)} \doteq catalog_{(2,2)}$) will be maximized if we have $item_{(1,1)} \doteq item_{(2,3)}$ and $item_{(1,2)} \doteq item_{(2,1)}$. That is, $\Re(catalog_{(1,1)}, catalog_{(2,2)})$ will be "0.636". Otherwise, $\Re(catalog_{(1,1)}, catalog_{(2,2)})$ will be "0.364". Therefore, the task in this step is to find *best matching configurations* that facilitate us to find best matching internal nodes.

The problem of finding *best matching configuration* is similar to the problem of finding *maximum weighted bipartite matching*. Hence, we are able to solve the problem of finding best matching configuration by using the algorithm for finding maximum weighted bipartite matching. In our implementation, we use the Hungarian method [7]. Note that we need to update the values of the attributes *Counter*, *Total*, and *Score* accordingly as initially their values are equal to "0". Intuitively, the value of attribute *Counter* is equal to $(p + \sum q_z)$, where p is the number of matching *non-inlined* leaf nodes, and q_z is the number of matching *inlined* leaf nodes retrieved from the temporary matching table. The value of attribute *Total* is equal to $(2k + \sum l_z)$, where k is the total number of *non-inlined* leaf nodes, and l_z the total number of *inlined* leaf nodes retrieved from the temporary matching table. The SQL query for updating the *Score* attribute is same as the one depicted in Figure 4(c).

2.4 Collecting Best Matching Internal Nodes

The result of the previous step is the best matching internal nodes stored 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. Lines 15-25 in Figure 3 depict the algorithm. The intuition behind this algorithm is to collect best matching internal nodes in top-down fashion, starting from the root nodes to the highest level of matching internal nodes. To retrieve the best matching internal nodes in each level, the algorithm invokes the *retriveMatching* algorithm (line 24, Figure 3).

Given an internal node x as the input, the *retriveMatching* algorithm works as follows. First, the algorithm checks whether node x is a root node, an *inlined* internal node, or a *non-inlined* internal node. If node x is a root node, then the algorithm executes the SQL query depicted in Figure 4(f). If node x is a *non-inlined* internal node, then the algorithm executes the SQL query depicted in Figure 4(g). The intuition behind the SQL query in Figure 4(g) is to find best matching node x in the M_x table whose parent nodes are best matching node in the `Matching` table. If node x is an *inlined* internal node, the algorithm executes the SQL query depicted in Figure 4(h). Given the temporary matching tables (Figures 5(b)-(c)) and the XML documents stored in RDBMS (Figure 2(a)), the *retriveMatching* algorithm results the `Matching` table as depicted in Figure 5(d). The semantics of the `Matching` table is depicted in Figures 6(a) and (b). The `Matching` table keeps the best matching internal nodes of two XML documents that will be used as the facilitator in detecting the changes (Phase 2).

3. DETECTING THE CHANGES

In section, we discuss how the changes are detected after the best matching subtrees are determined. We consider five types of changes as follows: *insertion of internal*

tempTb1 (DID1, DID2, PID1, PID2, Counter, Total, Score)

tempTb2 (DID1, DID2, PID1, PID2, ID1, ID2, Counter, Total, Score)

MATCHING (DID1, DID2, PID1, PID2, ID1, ID2, Score, Name)

INS_INT (DID1, DID2, PID, ID, Name)

DEL_INT (DID1, DID2, PID, ID, Name)

INS_LEAF (DID1, DID2, PID, ID, Name, Value)

DEL_LEAF (DID1, DID2, PID, ID, Name, Value)

UPD_LEAF (DID1, DID2, PID, ID, Name, Value1, Value2)

(a) Additional Tables

Attributes	Description
DID1	Document id of the first version
DID2	Document id of the second version
PID	Parent node id
PID1	Parent node id in the first version
PID2	Parent node id in the second version
ID	Node id
ID1	Node id in the first version
ID2	Node id in the second version
Counter	Number of matching nodes
Total	Total number of nodes
Score	Similarity score
Name	Node name
Value	Leaf node content
Value1	The old value of a leaf node
Value2	The new value of a leaf node

(b) Attributes and Descriptions

```
Input
  U : DTD of the XML documents
  the Matching table
  Two versions of an XML document stored in RDBMS
Output
  the delta tables

1  A ⟵ all the inlined leaf nodes in U
2  B ⟵ all the inlined internal nodes in U
3  C ⟵ all the non-inlined leaf nodes in U
4  D ⟵ all the non-inlined internal nodes in U
5  for all d in D do
6    nodeName ⟵ name(d); tbName ⟵ r_d;
7    detectInsertedInternalNodes(nodeName,tbName);
8    detectDeletedInternalNodes(nodeName,tbName);
9  end for
10 for all b in B do
11   parentNode ⟵ parent(b); parentTbName ⟵ r_parentNode;
12   parentNodeName ⟵ name(parentNode);
13   attrName ⟵ attribute(b); nodeName ⟵ name(b);
14   detectInsertedInternalNodes(nodeName,
         attrName,parentTbName,parentNodeName);
15   detectDeletedInternalNodes(nodeName,
         attrName, parentTbName,parentNodeName);
16 end for
17 for all a in A do
18   parentNode ⟵ parent(a); parentTbName ⟵ r_parentNode;
19   parentNodeName ⟵ name(parentNode);
20   nodeName ⟵ name(a); attrName ⟵ attribute(a);
21   detectInsertedLeafNodes(nodeName,
         attrName,parentTbName,parentNodeName);
22   detectDeletedLeafNodes(nodeName,
         attrName,parentTbName,parentNodeName);
23   detectUpdatedLeafNodes(nodeName,
         attrName,parentTbName,parentNodeName);
24 end for
25 for all c in C do
26   nodeName ⟵ name(c); parentNode ⟵ parent(c);
27   parentNodeName ⟵ name(parentNode); tbName ⟵ r_c;
28   detectInsertedLeafNodes(nodeName,tbName,parentNodeName);
29   detectDeletedLeafNodes(nodeName,tbName,parentNodeName);
30   detectUpdatedLeafNodes(nodeName,parentNodeName);
31   refineUpdatedLeafNodes(c);
32 end for
```

(c) The *changesDetector* Algorithm

Figure 6: Additional Tables, Their Attributes, The `changesDetector` Algorithm.

```
1  INSERT INTO INS_INT
2  SELECT doc_id1 AS DID1, doc_id2 AS DID2,
3     PID, ID, '[nodeName]' AS NAME
4  FROM [tbName] AS A
5  WHERE
6    A.DOC_ID = doc_id2 AND
7    NOT EXISTS (SELECT B.ID2
8       FROM MATCHING AS B
9       WHERE B.DID1 = doc_id1 AND
10         B.DID2 = doc_id2 AND
11         B.NAME = '[nodeName]' AND
12         B.ID2 = A.ID)
```

(a) Insertion of Internal Nodes (1)

```
1  INSERT INTO INS_INT
2  SELECT DISTINCT doc_id1 AS DID1,
3    doc_id2 AS DID2, I.ID AS PID,
4    NULL AS ID, '[nodeName]' AS NAME
5  FROM INS_INT AS I, [parentTbName] AS A
6  WHERE
7    I.DID1 = doc_id1 AND
8    I.DID2 = doc_id2 AND
9    I.NAME = '[parentNodeName]' AND
10   I.ID = A.ID AND
11   A.DOC_ID = doc_id2 AND
12   A.[attrName] IS NOT NULL
```

(c) Insertion of Internal Nodes (3)

```
1  INSERT INTO INS_LEAF
2  SELECT
3    doc_id1 AS DID1, doc_id2 AS DID2,
4    I.ID AS PID, NULL AS ID,
5    '[nodeName]' AS NAME,
6    A.[attrName] AS VALUE
7  FROM INS_INT AS I, [parentTbName] AS A
8  WHERE
9    I.DID1 = doc_id1 AND I.DID2 = doc_id2 AND
10   I.NAME = '[parentNodeName]' AND
11   I.ID = A.ID AND
12   A.DOC_ID = doc_id2 AND
13   A.[attrName] IS NOT NULL
```

(e) Insertion of Leaf Nodes (2)

```
1  INSERT INTO INS_LEAF
2  SELECT
3    doc_id1 AS DID1, doc_id2 AS DID2,
4    I.ID AS PID, A.ID AS ID,
5    '[nodeName]' AS NAME,
6    A.VAL AS VALUE
7  FROM INS_INT AS I, [tbName] AS A
8  WHERE
9    A.DOC_ID = doc_id2 AND
10   I.DID1 = doc_id1 AND
11   I.DID2 = doc_id2 AND
12   I.NAME = '[parentNodeName]' AND
13   I.ID = A.PID
```

(g) Insertion of Leaf Nodes (4)

```
1  INSERT INTO INS_INT
2  SELECT doc_id1 AS DID1,
3     doc_id2 AS DID2, I2.ID AS PID,
4     NULL AS ID, '[nodeName]' AS NAME
5  FROM [parentTbName] AS I1,
6       [parentTbName] AS I2
7  WHERE I1.DOC_ID = doc_id1 AND
8    I2.DOC_ID = doc_id2 AND
9    I1.[attrName] IS NULL AND
10   I2.[attrName] IS NOT NULL AND
11   EXISTS (SELECT ID1, ID2
12      FROM MATCHING AS B
13      WHERE B.ID1 = I1.ID AND
14        B.ID2 = I2.ID AND
15        B.DID1 = doc_id1 AND
16        B.DID2 = doc_id2 AND
17        B.NAME = '[parentNodeName]')
```

(b) Insertion of Internal Nodes (2)

```
1  INSERT INTO INS_LEAF
2  SELECT doc_id1 AS DID1, doc_id2 AS DID2,
3     I2.ID AS PID, NULL AS ID,
4     '[nodeName]' AS NAME,
5     I2.[attrName] AS VALUE
6  FROM [parentTbName] AS I1,
7       [parentTbName] AS I2
8  WHERE I1.DOC_ID = doc_id1 AND
9    I2.DOC_ID = doc_id2 AND
10   I1.[attrName] IS NULL AND
11   I2.[attrName] IS NOT NULL AND
12   EXISTS
13     (SELECT ID1, ID2 FROM MATCHING AS B
14      WHERE DID1 = doc_id1 AND
15        DID2 = doc_id2 AND
16        NAME = '[parentNodeName]' AND
17        B.ID1 = I1.ID AND B.ID2 = I2.ID)
```

(d) Insertion of Leaf Nodes (1)

```
1  INSERT INTO INS_LEAF
2  SELECT
3    doc_id1 AS DID1, doc_id2 AS DID2, A.PID, A.ID, '[nodeName]' AS NAME, A.VAL AS VALUE
4  FROM [tbName] AS A,
5    (SELECT M.DID1, M.DID2, M.PID1, M.PID2, VAL
6     FROM [tbName] AS A, MATCHING AS M
7     WHERE A.DOC_ID = doc_id2 AND  A.PID = M.PID2 AND M.NAME = '[parentNodeName]' AND
8       M.DID1 = doc_id1 AND M.DID2 = doc_id2
9     EXCEPT ALL
10    SELECT M.DID1, M.DID2, M.PID1, M.PID2, VAL
11    FROM [tbName] AS A, MATCHING AS M
12    WHERE A.DOC_ID = doc_id1 AND A.PID = M.PID1 AND M.NAME = '[parentNodeName]' AND
13      M.DID1 = doc_id1 AND M.DID2 = doc_id2 ) AS D
14 WHERE A.DOC_ID = doc_id2 AND A.PID = D.PID2 AND A.VAL = D.VAL
```

(f) Insertion of Leaf Nodes (3)

Figure 7: SQL queries (2).

nodes, *deletion of internal nodes*, *insertion of leaf nodes*, *deletion of leaf nodes*, and *content updates of leaf nodes*. The `changesDetector` algorithm for detecting these types of changes is depicted in Figure 6(c).

3.1 Insertion and Deletion of Internal Nodes

Intuitively, the *inserted internal node* is an internal nodes that is only available in the *new version* of XML documents. Similarly, the *deleted internal node* is an internal nodes that is only available in the *old version* of XML documents. This implies that the inserted and deleted internal nodes must not be best matching internal nodes as best matching internal nodes are available in both versions. The inserted/deleted *non-inlined* and *inlined* internal nodes are detected by using the `changesDetector` algorithm (Figure 6(c), lines 5-9 and lines 10-16 respectively).

Non-inlined Internal Nodes. For each *non-inlined* internal node d in DTD U, the `changesDetector` algorithm defines two parameters (line 6), and invokes the *detectInsertedInternalNodes* (line 7) and *detectDeletedInternalNodes* functions (line 8) for detecting inserted and deleted *non-inlined* internal nodes respectively.

The *detectInsertedInternalNodes* function executes the SQL query as depicted in Figure 7(a). Lines 7-12 are used to ensure that the inserted internal nodes are not best matching internal nodes. The *detectInsertedInternalNodes* function returns inserted internal nodes stored in the `INS_INT` table. For example, given the `Item` table (Figure 2(b)) and the `Matching` table (Figure 5(d)), the first row of `INS_INT` table depicted in Figure 5(e) is the result of the *detectInsertedInternalNodes* function.

The *detectDeletedInternalNodes* function executes the modified SQL query of the SQL query depicted in Figure 7(a). The `INS_INT` in line 1 is replaced by `DEL_INT`. The "*doc_id2*" (line 6) and "`ID2`" (lines 7 and 12) are replaced by "*doc_id1*" and "`ID1`" respectively. The *detectDeletedInternalNodes* function returns deleted internal nodes stored in the `DEL_INT` table. For example, given the `Item` table (Figure 2(b)) and the `Matching` table (Figure 5(d)), the first row of `DEL_INT` table depicted in Figure 5(f) is the result of the *detectDeletedInternalNodes* function.

Inlined Internal Nodes. For each *non-inlined* internal node b in DTD U, the `changesDetector` algorithm defines four parameters (lines 11-13). Next, the algorithm invokes the *detectInsertedInternalNodes* (line 14) and *detectDeletedInternalNodes* functions (line 15) for detecting inserted and deleted *inlined* internal nodes respectively.

The *detectInsertedInternalNodes* function executes the SQL queries as depicted in Figures 7(b) and (c). The SQL query in Figure 7(b) is used to detect inserted *inlined* internal nodes whose parent nodes are best matching internal nodes. Lines 9-10 are used to ensure that the inserted *inlined* internal nodes are only available in the new version. Lines 11-17 are used to ensure that the parent nodes of the inserted *inlined* internal nodes are best matching internal nodes. The SQL query depicted in Figure 7(c) is used to detect inserted *inlined* internal nodes whose parent nodes are inserted internal nodes. Line 10 is used to ensure that the parent nodes are inserted internal nodes. For example, given the `Item` table (Figure 2(b)) and the `Matching` table (Figure 5(d)), the second row of the `INS_INT` table depicted in Figure 5(e) is the result of the *detectInsertedInternalNodes* function.

To detect the deleted *inlined* internal nodes the *detectDeletedInternalNodes* function shall also execute two SQL queries derived from the SQL queries depicted in Figures 7(b) and (c). The "INS_INT" in line 1 (Figures 7(b) and (c)) is replaced by "DEL_INT". The SQL query depicted in Figure 7(b) is modified as follows. The "I2" (line 3), "IS NULL" (line 9), and "IS NOT NULL" (line 10) are replaced by "I1", "IS NOT NULL", and "IS NULL" respectively. We modify the SQL query depicted in Figure 7(c) as follows. The "INS_INT" (line 5) and "*doc_id2*" (line 11) are replaced by "DEL_INT" and "*doc_id1*" respectively. For example, given the Item table (Figure 2(b)) and the Matching table (Figure 5(d)), the second row of the DEL_INT table depicted in Figure 5(f) is the result of the *detectDeletedInternalNodes* function.

3.2 Insertion and Deletion of Leaf Nodes

Intuitively, the *inserted leaf node* is a leaf node that is only available in the new version of XML documents. We observe that the new leaf nodes should be either in the *best matching subtrees* or in the *inserted subtrees*. Similarly, the *deleted leaf node* is a leaf node that is only available in the old version of XML documents and should also be either in the *best matching subtrees* or in the *deleted subtrees*. If an inserted/deleted leaf node is in a best matching subtree, then the parent node of this leaf node must be a best matching internal node. If an inserted leaf node is in a newly inserted subtree, then the parent node of this leaf node must be an inserted internal node. If a deleted leaf node is in a deleted subtree, then the parent node of this leaf node must be a deleted internal node. The inserted/deleted *non-inlined* and *inlined* leaf nodes are detected by using the changesDetector algorithm (Figure 6(c)) in lines 17-24 and lines 25-32 respectively.

Inlined Leaf Nodes. For each *inlined* leaf node a in DTD U, the changesDetector algorithm defines four parameters (lines 18-20). Next, the algorithm invokes the *detectInsertedLeafNodes* (line 21) and *detectDeletedLeafNodes* functions (line 22) for detecting inserted and deleted *inlined* internal nodes respectively. Note that line 23 is used to detect updated leaf nodes that will be discussed in the later section.

The *detectInsertedLeafNodes* function executes two SQL queries as depicted in Figures 7(d) and (e). Figure 7(d) is used to detect inserted *inlined* leaf nodes that are in the best matching subtrees. Lines 10-11 are used to ensure that inserted leaf nodes are only available in the new version. Lines 12-17 are used to guarantee that the parent nodes of inserted leaf nodes are best matching internal nodes. To detect inserted *inlined* leaf nodes that are in the newly inserted subtree we use SQL query depicted in Figure 7(e). Lines 12-13 are used to indicate that inserted leaf nodes must be only available in the new version. Lines 10-11 are used to make sure that the parent nodes are inserted internal nodes. Given the Item table (Figure 2(b)) and the Matching table (Figure 5(d)), the tuples marked by "#" in the INS_LEAF table depicted in Figure 5(g) are the result of the *detectInsertedLeafNodes* function.

The *detectDeletedLeafNodes* function executes two modified SQL queries of the SQL queries depicted in Figures 7(d) and (e). The "INS_LEAF" in line 1 (Figures 7(d) and (e)) is replaced by the "DEL_LEAF". The SQL query in Figure 7(d) is modified as follows. The "I2" in lines 3 and 5 is replaced by "I1". The "IS NULL" (line 10) and "IS NOT NULL" (line 11) are replaced by "IS NOT NULL" and "IS NULL" respectively. The SQL query in Figure 7(e) is modified as follows. The "INS_INT" (line 7) and "*doc_id2*" (line 12) are replaced by "DEL_INT" and "doc_id1" respectively. Given the Item table (Figure 2(b)) and the Matching table (Figure 5(d)), the tuples marked by "#" in the DEL_LEAF table in Figure 5(h) are the result of the *detectDeletedLeafNodes* function.

Non-inlined Leaf Nodes. For each *non-inlined* leaf node c in DTD U, the changesDetector algorithm defines three parameters (lines 26-27). Next, the changesDetector algorithm invokes the *detectInsertedLeafNodes* (line 29) and *detectDeletedLeafNodes* functions (line 29) for detecting the inserted and deleted *non-inlined* leaf nodes respectively. Note that line 30 (Figure 6(c)) is used to detect the updated leaf nodes that will be discussed in the later section.

The *detectInsertedLeafNodes* function shall execute the SQL queries as depicted in Figures 7(f) and (g). Figure 7(f) is used to detect inserted *non-inlined* leaf nodes that are in the best matching subtrees. Lines 5-8 and lines 10-13 are used to find the *non-inlined* leaf nodes that are in the new and old versions respectively. Operator "EXCEPT ALL" in line 9 is used to find non-inlined leaf nodes that are only available in the new version. Figure 7(g) is used to detect inserted *non-inlined* leaf nodes that are in the deleted subtrees. Lines 12-13 are used to guarantee that the parent nodes of the leaf nodes are inserted internal nodes. Given the Info table (Figure 2(b)) and the Matching table (Figure 5(d)), the tuples in the INS_LEAF table depicted in Figure 5(g) that are not marked with "#" are the result of the *detectInsertedLeafNodes* function.

The *detectDeletedLeafNodes* function executes the SQL queries as depicted in Figures 7(f) and (g) after slightly modifications. The "INS_LEAF" in line 1 (Figures 7(f) and (g)) is replaced by the "DEL_LEAF". The "*doc_id2*" in lines 7 and 14 (Figure 7(f)) and line 9 (Figure 7(g)) is replaced by "*doc_id1*". We replace the "*doc_id1*" in line 12 (Figure 7(f)) with "*doc_id2*". The "PID2" in lines 7 and 14 (Figure 7(f)) is replaced by "PID1". We replace the "PID1" in line 12 (Figure 7(f)) with "PID2". Given the Info table (Figure 2(b)) and the Matching table (Figure 5(d)), the tuples in the DEL_LEAF table depicted in Figure 5(h) that are not marked with "#" are the result of the *detectDeletedLeafNodes* function. The *detectInsertedLeafNodes* and *detectDeletedLeafNodes* functions return the updated *non-inlined* leaf nodes as the updated *non-inlined* leaf nodes can be decomposed into pairs of deleted and inserted leaf nodes. The highlighted tuples in Figures 5(g) and (h) are the updated *non-inlined* leaf nodes detected as inserted and deleted leaf nodes respectively.

3.3 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. In addition to this, the parent nodes of the updated leaf nodes must be the best matching internal nodes. In the changesDetector algorithm, the updated leaf nodes are detected after the inserted and deleted leaf nodes are detected.

Inlined Leaf Nodes. The *detectUpdatedLeafNodes* function executes the SQL query as depicted in Figure 8(a). Lines 10-12 are used to ensure that the updated leaf nodes are available in both versions (lines 10-11) and they have different values (line 12). For example, given the Item table (Figure 2(b)) and the Matching table (Figure 5(d)), the *detectUpdatedLeafNodes* function shall result the first tuple of the UPD_LEAF table as depicted in Figure 5(i).

Non-inlined Leaf Nodes. The *detectUpdatedLeafNodes* function shall execute the SQL query depicted in Figure 8(b). We notice that we join three tables, namely, the DEL_LEAF,

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

(a) Update of Leaf Nodes (1)

```
1  INSERT INTO UPD_LEAF
2  SELECT DISTINCT
3    doc_id1 AS DID1, doc_id2 AS DID2,
4    D.PID AS PID1, I.PID AS PID2,
5    D.ID AS ID1, I.ID AS ID2,
6    '[nodeName]' AS NAME,
7    D.VALUE AS VALUE1, I.VALUE AS VALUE2
8  FROM DEL_LEAF AS D, INS_LEAF AS I, MATCHING AS M
9  WHERE
10   M.DID1 = doc_id1 AND M.DID2 = doc_id2 AND
11   D.DID1 = doc_id1 AND D.DID2 = doc_id2 AND
12   I.DID1 = doc_id1 AND I.DID2 = doc_id2 AND
13   M.PID1 = D.PID AND M.PID2 = I.PID AND
14   D.NAME = I.NAME AND D.VALUE != I.VALUE AND
15   I.NAME = '[nodeName]' AND
16   M.NAME = '[parentNodeName]'
```

(b) Update of Leaf Nodes (2)

Dataset Code	Nodes	Filesize (KB)
Sigmod-01	331	13
Sigmod-02	554	21
Sigmod-03	890	34
Sigmod-04	1,826	70
Sigmod-05	2,718	104
Sigmod-06	4,717	180
Sigmod-07	8,794	337
Sigmod-08	18,866	721
Sigmod-09	37,725	1,444
Sigmod-10	89,323	3,431

(c) Sigmod Dataset

Dataset Code	Nodes	Filesize (KB)
Univ-01	485	36
Univ-02	930	68
Univ-03	2,298	167
Univ-04	4,577	334
Univ-05	6,785	493
Univ-06	9,057	664
Univ-07	18,154	1,319
Univ-08	45,264	3,294

(d) University Dataset

```
<!ELEMENT univ (school)* >
<!ELEMENT school
    (sname,departments) >
<!ELEMENT departments (department)* >
<!ELEMENT department (dname,courses) >
<!ELEMENT courses (course)* >
<!ELEMENT course(#PCDATA) >
<!ELEMENT sname (#PCDATA) >
<!ELEMENT dname (#PCDATA) >
```

(e) University DTD

Figure 8: SQL Queries for Detecting Updated Leaf Nodes and Data sets.

INS_LEAF, and Matching tables. This is because the updated *non-inlined* leaf nodes are already decomposed into pairs of deleted and inserted *non-inlined* leaf nodes stored in DEL_LEAF and INS_LEAF respectively. Line 13 is used to guarantee that the parent nodes of the deleted and inserted leaf nodes are the best matching internal nodes. The updated leaf nodes must have the same node name, but different values (line 14).

We observed that the *detectUpdatedLeafNodes* function for detecting updated *non-inlined* leaf nodes may return *incorrect* results in some conditions as follows. First, there are more than one updated *non-inlined* leaf nodes under the same parent nodes. Second, there are deletion/insertion and update of *non-inlined* leaf nodes occurred under the same parent nodes. In our example, we have node with value "*Extra 1 year warranty*" deleted and node with value "*Series 170S6FG*" updated. These nodes are under the same parent node. The SQL query depicted in Figure 8(b) returns the last two rows of the UPD_LEAF table as depicted in Figure 5(i). Therefore, we use the *refineUpdatedLeafNodes* function to correct the result of the *detectUpdatedLeafNodes* function. The *refineUpdatedLeafNodes* algorithm is similar to the one in [6].

For example, given the DEL_LEAF table (Figure 5(h)), the INS_LEAF table (Figure 5(g)), and the Matching table (Figure 5(d)), the *detectUpdatedLeafNodes* function shall result the UPD_LEAF table as depicted in Figure 5(i) (the last two tuples). After the changesDetector algorithm invokes *refineUpdatedLeafNodes* function, the highlighted row of the UPD_LEAF table as depicted in Figure 5(i) is deleted.

4. EXPERIMENTAL RESULTS

We have implemented HELIOS entirely in Java. The Java implementation and the database engine were run on a Microsoft Windows 2000 Professional machine having Pentium 4 1.7 GHz processor with 512 MB of memory. The database system was IBM DB2 UDB 8.1. Appropriate indexes on the relations are created. We used a set of synthetic XML documents based on SIGMOD DTD[1] and University DTD (Figure 8(e)). The characteristics of the datasets are depicted in Figure 8(c) and (d). 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. We compare HELIOS with XANDY [6] and X-Diff [10][2]. Note that we focus on the number of nodes in the datasets as the higher the number of nodes the database engine will join more number of tuples. Figure 9(a) depicts the comparison of the execution time of HELIOS, XANDY, and X-Diff for different file size while we fix the number of nodes to 890 nodes and the percentage of changes to 3%. The performances of HELIOS, XANDY, and X-Diff are slightly affected by the increments of the file size. We shall see in the later discussion that the increments of the number of nodes have more influences on the performances of these approaches.

Execution Time vs Number of Nodes. In this set of experiments, we study the performance of all approaches for different number of nodes. The percentage of changes is set to "3%" and "9%". The threshold θ is set to "0.0" which shall give us the upper bound of the execution time.

Sigmod Data Sets. Figure 9(c) depicts the comparison of the execution time of XANDY and HELIOS for the first phase (3% changes). HELIOS is 3 times faster than XANDY in average. Figure 9(d) depicts the comparison of the execution time of different approaches for the change detection phase (3% changes). HELIOS performs better than XANDY except for the smallest data set. HELIOS is 60 times faster than XANDY in average. We notice that the difference of execution time between HELIOS and XANDY increases as the number of nodes increases. Figure 9(e) and (f) depict the overall performance of each approach when the percentage of change is set to 3% and 9% respectively. X-Diff performs better than XANDY for the first three data sets, and HELIOS for the first two data sets. HELIOS is 3.3 times faster than XANDY in average. HELIOS is faster than XANDY for the following reasons. As HELIOS uses the Shared-Inlining schema for storing the XML documents, the leaf and internal nodes are shredded into several tables. XANDY uses SUCXENT schema in which the leaf and internal nodes are stored in the LeafValue and AncestorInfo tables respectively. Therefore, there are more tuples to be joined by the SQL queries issued by XANDY. Note that X-Diff is unable to detect the changes on the XML documents that have number of nodes over 5000 nodes due to lack of main memory.

In the next set of experiments, we examine the sub phases in first and second phases. We use the first five data sets. The percentage of changes is set to "3%" and the threshold θ is set to "0.0". Figure 9(g) shows the sub phases of the first phase in HELIOS. We observe that the execution time of the first phase is mostly taken by the execution time of finding the matching *article* nodes (around 54.23%). Figure 9(h) shows the sub phases of the second phase in HELIOS. The total execution time of this second phase is mostly taken by the execution time of detecting updates (around 30.2%), of detecting deletion of leaf node (around 28.5%), and of detecting insertion of leaf node (around 27.6%).

University Data Sets. Figure 9(i) depicts the overall performance of HELIOS and XANDY. We set the percentage of the changes to "3%". We observe that HELIOS is up to 6.2 times faster that XANDY.

Next, we vary the number of *inlined* leaf nodes. We increase the number of *inlined* leaf nodes up to eight additional *inlined* leaf nodes while we fix the numbers of nodes in the first versions of XML documents to 1175 nodes. Figure 9(j) depicts the performance of HELIOS and XANDY for different number of *inlined* leaf nodes. We observed that the performances of HELIOS and XANDY are influenced by number of

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

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

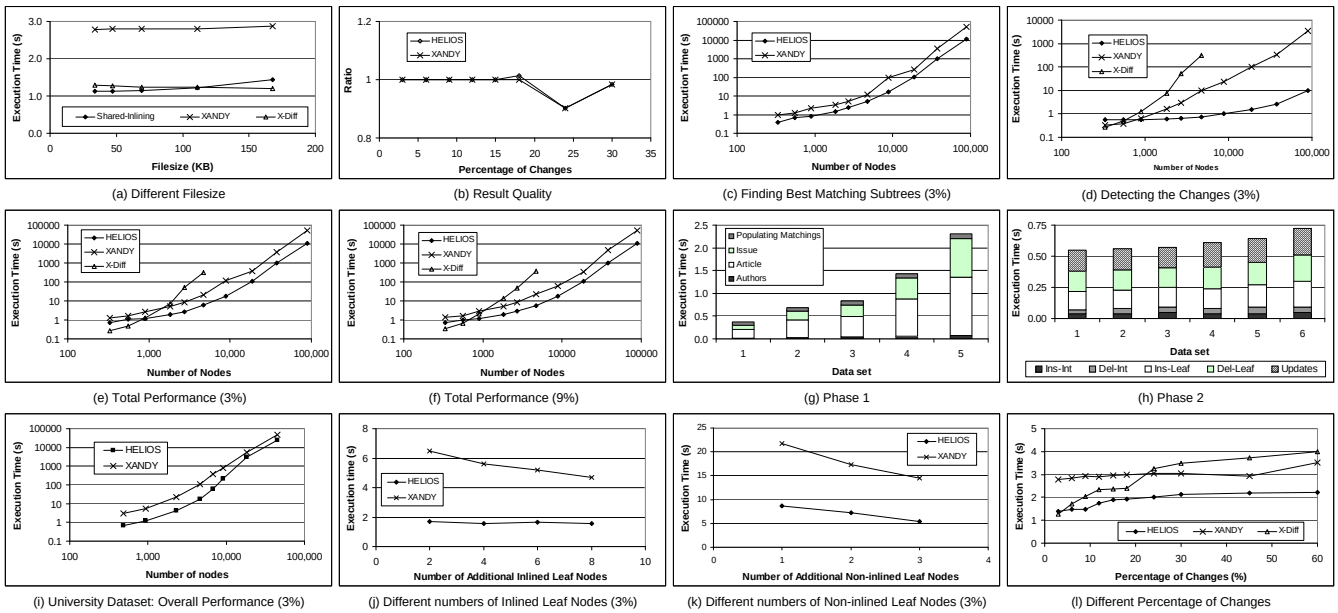


Figure 9: Experimental Results.

inlined leaf nodes. This is because we have less number of subtrees to be matched as we increase the number of *inlined* leaf nodes and fix the total number of nodes.

In the next experiments, we vary the number of *non-inlined* leaf nodes. We increase the number of *non-inlined* leaf nodes up to three additional *inlined* leaf nodes while the numbers of nodes in the first versions of XML documents are fixed to 2345 nodes. Figure 9(k) depicts the performance of HELIOS and XANDY for different number of *non-inlined* leaf nodes. We observed that both approaches are influenced by number of *non-inlined* leaf nodes. In HELIOS, we shall have more relations for storing the *non-inlined* leaf nodes. The size of each relation will be smaller. That is, we have less number of tuples to join for each SQL query.

In the next set of experiments, we examine the effects of the percentages of changes to the performance of each approach. We use dataset "SIGMOD-03". Figure 9(l) shows the performance of each approach when we vary the percentages of changes. We notice that the percentage of changes slightly affects the performances of HELIOS and XANDY. The performance of X-Diff is affected by the percentage of changes.

Result Quality. In this set of experiments, we examine the result quality of HELIOS and XANDY by using the "`UNIV-01`" dataset. We vary the percentage of change from 3% up to 30%. We calculate the ratio $R = x/y$, where x is the *number of changed nodes* in result delta of HELIOS and XANDY, and y is the *number of changed nodes* in the result delta of X-Diff. The ratios are plotted in Figure 9(b). We observed that XANDY is able to detect the optimal or near optimal deltas. We observed that XANDY detects the same deltas as X-Diff until the percentage of the changes reaches 18%. HELIOS detects the same deltas as X-Diff until the percentage of the changes reaches 15% The quality ratios of X-Diff and XANDY, and of X-Diff and HELIOS are smaller than 1 when the percentage of the changes is larger than 20%. This happens because X-Diff detects a deletion and insertion of subtrees as a set of update operations.

5. CONCLUSIONS

In this paper, we present a novel relational approach for detecting the changes on unordered XML documents using a schema-conscious approach (called HELIOS). This paper is motivated by the fact that a growing body of work suggests that schema-conscious approaches perform better than schema-oblivious approaches as far as XML query processing is concerned. The characteristics of schema-conscious approach raise certain challenges. For instance, the underlying relational schema is DTD-dependent. To address the challenges, we present a general framework that is able to detect all changes accurately independent of the changes to the underlying relational schema due to different structure of XML documents. We compare HELIOS to XANDY and X-Diff. The experimental results show that HELIOS is up to 6.7 times faster than XANDY, and up to 52 times faster than X-Diff. The result quality of HELIOS is comparable to XANDY. As parts of our future work, we would like to extend our framework so that it can handle recursive DTDs.

6. REFERENCES

[1] S. CHAWATHE, H. GARCIA-MOLINA. Meaningful Change Detection in Structured Data. *In SIGMOD*, 1997.

[2] S. CHAWATHE, A. RAJARAMAN, H. GARCIA-MOLINA, J. WIDOM. Change Detection in Hierarchically Structured Information. *In SIGMOD*, 1996.

[3] Y. CHEN, S. MADRIA, S. S. BHOWMICK. DiffXML: Change Detection in XML Data. *In DASFAA*, Korea, 2004.

[4] G. COBENA, S. ABITEBOUL, A. MARIAN. Detecting Changes in XML Documents. *In ICDE*, San Jose, 2002.

[5] R. KRISHNAMURTHY, V. T. CHAKARAVARTHY, R. KAUSHIK, J. F. NAUGHTON. Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation.*In Proc. of IEEE ICDE*, 2004.

[6] E. LEONARDI, S. S. BHOWMICK, S. MADRIA. XANDY: Detecting Changes on Large Unordered XML Documents Using Relational Databases. *In DASFAA*, China, 2005.

[7] C. PAPADIMITRIOU, K. STEIGLITZ. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[8] 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.

[9] F. TIAN, D. DEWITT, J. CHEN AND C. ZHANG. The Design and Performance Evaluation of Alternative XML Storage Strategies. *ACM Sigmod Record, Vol. 31(1), 2002* .

[10] Y. WANG, D. J. DEWITT, J. CAI. X-Diff: An Effective Change Detection Algorithm for XML Documents. *In ICDE*, Bangalore, 2003.