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 have 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.


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.

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.
In this paper, we present a novel relational approach for detecting the changes in unordered XML documents called HELIOS (sCHEma-conscious xml-Enabled change detectIOn System) 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.

Figure 2: XML Trees and XML in RDBMS.

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_{n1}, r_{n2}, ..., r_{nk}}, 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
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 $\ell_v$ be a leaf node in the version $v$ of an XML document, where $\ell_v \in L_v$. The textual content of $\ell_v$ 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 $\ell_1$ and $\ell_2$ be two leaf nodes in the first and second versions of an XML tree respectively. Then, $\ell_1$ and $\ell_2$ are matching leaf nodes (denoted as $\ell_1 \leftrightarrow \ell_2$) if name($\ell_1$) = name($\ell_2$), level($\ell_1$) = level($\ell_2$), and value($\ell_1$) = value($\ell_2$). 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 $n_{(p)}$ be a leaf node $n$ stored in the $root$ table with doc_id=$p$ and id=$q$. We notice that there are four matching leaf nodes, namely, $info(1,2) \leftrightarrow info(2,7)$, $info(0,4) \leftrightarrow info(2,1)$, $info(1,5) \leftrightarrow info(2,2)$, 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 $G_{1(pid_1)}$ and $G_{2(pid_2)}$ be two sets of non-inlined leaf nodes in the first and second versions of an XML document respectively, where $\forall \ell_1 \in G_1$ have the parent node id $pid_1$, and $\forall \ell_2 \in G_2$ 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 $\exists \ell_1 \leftrightarrow \ell_2$ where $\ell_1 \in G_{1(pid_1)}$ and $\ell_2 \in G_{2(pid_2)}$. 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 $\mathcal{R}$ of two matching groups $G_1$ and $G_2$ is defined as follows: $\mathcal{R}(G_1, G_2) = \frac{|G_1 \cap G_2|}{|G_1 \cup G_2|}$, 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 $|G_{(1,1)} \cup G_{(2,3)}| = 5$.

Given two versions of an XML document stored 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 $tempTbl$ 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. 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).
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 \( i_1 \cong i_2 \)) if the following conditions are satisfied: 1) \( \text{name}(i_1) = \text{name}(i_2) \), 2) \( \text{level}(i_1) = \text{level}(i_2) \), and 3) \( \exists \ell_1, \exists \ell_2 \) such that \( \ell_1 \Leftarrow \ell_2 \), where \( i_1 = \text{ancestor}(\ell_1) \) and \( i_2 = \text{ancestor}(\ell_2) \).

For each non-inlined internal node \( a_x \), the bestMatchingFinder 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 \( M_a \), that is the instance of the tempTb2 table as depicted in Figures 6(a) and (b).

If node \( a_x \) 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 (\( i_1 \cong i_2 \)) if \( \exists a_1, \exists a_2 \) such that \( \text{value}(a_1) = \text{value}(a_2) \), where \( a_1 = \text{child}(i_1) \) and \( a_2 = \text{child}(i_2) \), and \( \text{attribute}(a_1) = \text{attribute}(a_2) \). 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 \( M_a \).

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 \( \forall z \in Z \) such that \( i \cong z \), \( i \) and \( z \) are best matching internal nodes (denoted by \( i \cong z \)) iff \( (R(i, z_i) > R(i, z_i)) \land (0 < p \leq |Z|) \land q \neq p \). For example, in the M_Item table we notice that item\(_{(1,2)}\) table that is used to replace a part of the SQL query. The similarity score of matching catalog nodes \( \text{catalog}(i_1, i_2) \) will be maximized if we have item\(_{(1,1)}\) \cong item\(_{(2,1)}\) and item\(_{(1,2)}\) \cong item\(_{(2,2)}\). That is, \( R(\text{catalog}(i_1), \text{catalog}(i_2)) \) will be “0.636”. Otherwise, \( R(\text{catalog}(i_1), \text{catalog}(i_2)) \) 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 \( p + \sum q_i \), where \( p \) is the number of matching non-inlined leaf nodes, and \( q_i \) is the number of matching inlined leaf nodes retrieved from the temporary matching table. The value of attribute Total is equal to \( 2k + \sum l_i \), where \( k \) is the total number of non-inlined leaf nodes, and \( l_i \) 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 retrieveMatching algorithm (line 24, Figure 3).

Given an internal node \( x \) as the input, the retrieveMatching 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 \( M_a \) 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 retrieveMatching 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
nodes, deletion of internal nodes, insertion of leaf nodes, and content updates of leaf nodes. The changesDetector algorithm detects 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 node that is only available in the new version of XML documents. Similarly, the deleted internal node is an internal node 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 the 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 detectInsertedInternalNodes 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 the DEL_INT table depicted in Figure 5(f) is the result of the detectInsertedInternalNodes function.

#### Inlined Internal Nodes

For each non-inlined internal node d 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 "11", "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 only available in the new version. 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 two 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,
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 node. 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 DTD1 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 SUCCENT 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 than 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
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