Xandy: Detecting Changes on Large Unordered XML Documents Using Relational Databases

Erwin Leonardi[1] Sourav S. Bhowmick[1] Sanjay Madria[2]

School of Computer Engineering[1]
Nanyang Technological University
Singapore
{pk909134,assourav}@ntu.edu.sg

Department of Computer Science[2]
University of Missouri-Rolla
Rolla, MO 65409
madrias@umr.edu

Abstract. Previous works in change detection on XML documents are not suitable for detecting the changes to large XML documents as it requires a lot of memory to keep the two versions of XML documents in the memory. In this paper, we take a more conservative yet novel approach of using traditional relational database engines for detecting the changes to large *unordered* XML documents. We elaborate how we detect the changes on unordered XML documents by using relational database. To this end, we have implemented a prototype system called Xandy that converts XML documents into relational tuples and detects the changes from these tuples by using SQL queries. Our experimental results show that the relational approach has better scalability compared to published algorithms like X-Diff. The result quality of our approach is comparable to the one of X-Diff.

1 Introduction

Detecting changes to XML data is an important research problem. Cobena et al. [3] proposed an algorithm, called XyDiff, for detecting changes on ordered XML documents by using the signature and weight of nodes. XMLTreeDiff [2] is also proposed for solving the problem of detecting changes for ordered XML documents by using DOMHash. In [10], the authors presented X-Diff, an algorithm for detecting the changes on unordered XML documents. In this paper, we focus on detecting the changes on the unordered XML documents.

The changes on unordered XML documents can be classified into two types: *changes to the internal nodes* and *changes to the leaf nodes*. An *internal node* does not contain textual data. For example, consider the two versions of an XML document in Figure 1. Nodes 2 and 7 in Figure 1(a) are the internal nodes. The changes that occur in the internal nodes are called as *structural changes* as they modify the structure but do not change the textual data content. There are two types of *structural changes* for unordered XML documents: *insertion of internal nodes*, and *deletion of internal nodes*. For instance, node 102 in Figure 1(b) is an example of internal nodes insertion. A *leaf node* is the node/attribute which contains textual data. For example, node 3 is a leaf node which has name "category" and textual content "Memory". The changes in the leaf nodes are called *content changes* as they modify the textual data content. There are three

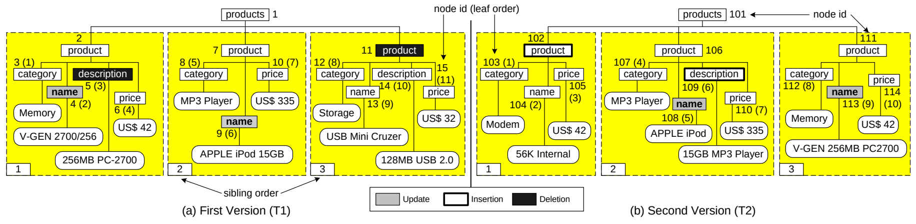


Fig. 1. Example.

types of *content changes* for unordered XML documents: *insertion of leaf nodes*, *deletions of leaf nodes*, and *content update of a leaf nodes*. For example, a leaf node 5 is a deleted leaf node. In this paper, we present a novel technique for detecting the *content* and *structural* changes in *unordered* XML using RDBMS.

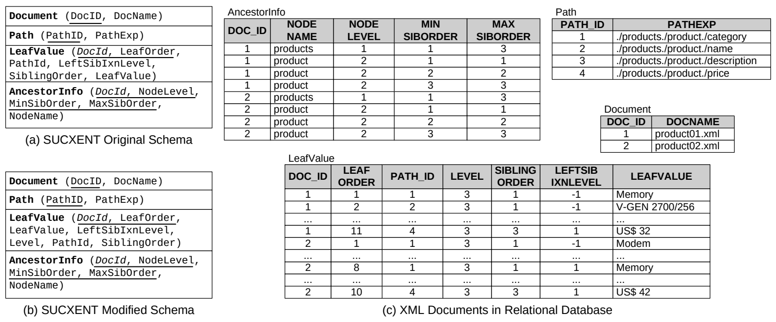
The main-memory-based approaches have some limitations as far as change detection is concerned. First, they require the entire trees (i.e., DOM trees) of both versions of an XML document to be memory resident. This problem is exacerbated by the fact that these trees are typically much larger than their XML documents. Thus, the scheme is not scalable for very large XML documents. In fact, the scheme is inefficient. We need to parse an XML document multiple times whenever we want to compare it with more than one document at different times.

The above limitations coupled with the recent success in storing XML data in relational databases [4, 8, 7] force us to ask whether we can address these problems by using relational techniques to detect the changes on XML documents. In our preliminary effort in [5, 1], we have demonstrated that it is indeed possible to use the relational database to detect the changes to *ordered* XML data. In [5, 1], we present relational approaches for detecting the *content changes* on *ordered* XML documents. However, the underlying relational schema of [1] is simplistic and is not efficient for path expressions query processing. Ideally, a change detection system build on top of a relational database should also support efficient insertion and extraction of XML documents and efficient execution of path expression queries. Hence, our approach in [5] uses `SUCXENT` schema that enables us to insert, extract, and query XML data efficiently [7].

In this paper, we present a novel relational approach for detecting the changes on *unordered* XML documents called XANDY (**X**ml en**A**bled cha**N**ge **D**etection s**Y**stem). Our approach differs from our previous efforts in two ways. First, we focus on unordered XML documents. To the best of our knowledge, currently, there is no published approach for detecting the change on *unordered* XML documents by using relational database. Detecting changes on *unordered* trees are substantially harder than that on *ordered* trees [10]. Second, we detect both content and structural changes.

2 Background

In this section, we present the relational database scheme used for storing two versions of XML documents. We have extended the relational schema of our XML storage system called `SUCXENT` (**S**chema **U**n**C**oncious **X**ML **EN**abled Sys**T**em) [7]. We chose `SUCXENT` because we have shown in [7] that our approach outperforms significantly the current state-of-the-art model mapping approaches



Document (DocID, DocName)

Path (PathID, PathExp)

LeafValue (*DocId*, LeafOrder, PathId, LeftSibIxnLevel, SiblingOrder, LeafValue)

AncestorInfo (*DocId*, NodeLevel, MinSibOrder, MaxSibOrder, NodeName)

(a) SUCXENT Original Schema

Document (DocID, DocName)

Path (PathID, PathExp)

LeafValue (*DocId*, LeafOrder, LeafValue, LeftSibIxnLevel, Level, PathId, SiblingOrder)

AncestorInfo (*DocId*, NodeLevel, MinSibOrder, MaxSibOrder, NodeName)

(b) SUCXENT Modified Schema

AncestorInfo

DOC_ID	NODE NAME	NODE LEVEL	MIN SIBORDER	MAX SIBORDER
1	products	1	1	3
1	product	2	1	1
1	product	2	2	2
1	product	2	3	3
2	products	1	1	3
2	product	2	1	1
2	product	2	2	2
2	product	2	3	3

Path

PATH_ID	PATHEXP
1	./products./product./category
2	./products./product./name
3	./products./product./description
4	./products./product./price

Document

DOC_ID	DOCNAME
1	product01.xml
2	product02.xml

LeafValue

DOC_ID	LEAF ORDER	PATH_ID	LEVEL	SIBLING ORDER	LEFTSIB IXNLEVEL	LEAFVALUE
1	1	1	3	1	-1	Memory
1	2	2	3	1	-1	V-GEN 2700/256
...	...	...	...	...	...	...
1	11	4	3	3	1	US$ 32
2	1	1	3	1	-1	Modem
...	...	...	...	...	...	...
2	8	1	3	1	1	Memory
...	...	...	...	...	...	...
2	10	4	3	3	1	US$ 42

(c) XML Documents in Relational Database

Fig. 2. XML Documents in Relational Database.

like XParent [4] as far as storage size, insertion time, extraction time, and path expression queries are concerned [1]. The `SUCXENT` schema is shown in Figure 2(a). We use the `Document` table for storing the names of the documents in the database. This allows us to store multiple versions of XML documents. The `Path` table is used to record the all paths from the root to the leaf nodes. It maintains the path ids and the relative path expressions as instances of the `PathID` and `PathExp` attributes respectively.

The `LeafValue` table is used for storing the information of the leaf nodes. The `DocID` attribute indicates which XML document a particular leaf node belongs to. The `PathID` attribute maintains the id of the path of a particular leaf node stored in the `Path` table. The `LeafOrder` attribute is used to record the *node order* of the leaf nodes in an XML tree. For example, consider the XML tree in Figure 1(a). When we parse the XML document, we will find the leaf node "category" with value "Memory" as the first leaf node in the document. Hence, we assign the `LeafOrder` equal to "1" for this leaf node. The `LeafOrder` of the next leaf node (node "name" with value "V-GEN 2700/256") is equal to "2". Two leaf nodes have the same `SiblingOrder` if they share the same parent. For example, the leaf nodes with `LeafOrder` equal to "1", "2", "3", and "4" shall have the same `SiblingOrder` (equal to "1") since they share the same parent node (node 2). The dotted boxes in Figure 1 indicate the leaf nodes that have the same `SiblingOrder`. The `LeftSibIxnLevel` (Left Sibling Intersection Level) is the level at which the leaf nodes belonging to a particular sibling order intersect the leaf nodes belonging to the sibling order that comes immediately before. For example, consider the leaf nodes with `SiblingOrder` equal to "2" in the XML tree. These leaf nodes shall intersect with the leaf nodes having `SiblingOrder` equal to "1" at the node "products" (id=1) which is at level 1. The `LeafValue` stores the textual content of the leaf nodes. Note that the attribute `LeftSibIxnLevel` in this table is only useful for constructing the XML documents from the relational database [7]. We use the `AncestorInfo` table for storing the information of the internal nodes. The `DocID` attribute indicates which XML document a particular ancestor node belongs to. We record the names and the level of ancestor nodes in the `NodeName` and `NodeLevel` attributes respectively. The `MinSibOrder` and

[1] Jiang et al. has shown in [4] that XParent outperforms various existing model mapping approaches.

`MaxSibOrder` store the minimum and maximum sibling orders of the leaf nodes under a particular ancestor node respectively. For example, the node "products" (id=1) in Figure 1(a) has `MinSibOrder` and `MaxSibOrder` equal to "1" and "3" respectively. Node "product" (id=7) has `MinSibOrder` and `MaxSibOrder` equal to "2" and "2" respectively.

For detecting the changes in unordered XML documents, we need to modify the `SUCXENT` schema. We add the attribute `Level` in the `LeafValue` table to store the level of the leaf nodes. The modified `SUCXENT` schema is depicted in Figure 2(b). Figure 2(c) shows the tables containing the two shredded XML documents in Figure 1 (partial view only).

3 Phase 1: Finding The Best Matching Subtrees

Suppose we have two versions of an XML tree, T_1 and T_2. The objective of this phase is to find the most similar subtrees in T_1 and T_2. First, the algorithm determines the *matching leaf nodes* in in T_1 and T_2 by issuing a SQL query against the database. Then it starts to match the ancestor nodes of the matching leaf nodes up to the root nodes. Note that the algorithm issues several SQL queries to match the subtrees. The most similar subtrees are considered as the *best matching subtrees*. This first phase results a set of *best matching internal nodes* at which the best matching subtrees are rooted. We use the information of the best matching subtrees to determine the *minimum delta*. In this section, we shall elaborate this phase further.

Definition 1. [**Matching Leaf Nodes**] *Let $L(T_1)$ and $L(T_2)$ be two sets of the leaf nodes in T_1 and T_2 respectively. Let name(ℓ), level(ℓ), and value(ℓ) be the node name, node level, and textual content of a leaf node ℓ respectively. Then ℓ_1 and ℓ_2 are **matching leaf nodes** (denoted as $\ell_1 \leftrightarrow \ell_2$) if name($\ell_1$) = name($\ell_2$), level($\ell_1$) = level($\ell_2$), and value($\ell_1$) = value($\ell_2$), where $\ell_1 \in L(T_1)$ and $\ell_2 \in L(T_2)$.*

Next, we define the notion of *matching sibling orders*. A set of leaf nodes that have the same parent node will have the same sibling order. The matching sibling orders can summarize the information of matching leaf nodes. Hence, the storage space needed for storing the matching information is reduced.

Definition 2. [**Matching Sibling Orders**] *Let so_1 and so_2 be two sibling orders in T_1 and T_2 respectively. Let $P = \{p_1, p_2, ..., p_x\}$ and $Q = \{q_1, q_2, ..., q_y\}$ be two sets of leaf nodes, where $\forall p_i \in P$ have the same sibling order so_1, and $\forall q_j \in Q$ have the same sibling order so_2. Then so_1 and so_2 are the **matching sibling orders** (denoted by $so_1 \Leftrightarrow so_2$) if $\exists p_i$ $\exists q_j$ such that $p_i \leftrightarrow q_j$ where $p_i \in P$ and $q_j \in Q$.*

After determining the matching sibling orders, we are able to find the *possible matching internal nodes* at which the *possible matching subtrees* are rooted. Informally, the *possible matching subtrees* are the subtrees in which they have at least one matching sibling orders. Note that the subtrees in T_1 are possible to be matched to more than one subtrees in T_2.

Definition 3. [**Possible Matching Subtrees**] *Let $I(T_1)$ and $I(T_2)$ be two sets of the internal nodes in T_1 and T_2 respectively. Let S_1 and S_2 be two subtrees*

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

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

(a) Algorithm *findBestMatchingSubtree*

```
SELECT
   D.LEVEL, D.SO1, D.SO2, D.COUNTER*2 AS COUNTER,
   V1.TOTAL+V2.TOTAL AS TOTAL
FROM
  (SELECT T.LEVEL, T.SO1, T.SO2, COUNT(T.SO1) AS COUNTER
   FROM
     (SELECT
        L1.LEVEL, L1.LEAFORDER AS LO1, L2.LEAFORDER AS LO2,
        L1.PATH_ID, L1.LEAFVALUE,
        L1.SIBLINGORDER AS SO1, L2.SIBLINGORDER AS SO2
      FROM
        LEAFVALUE AS L1, LEAFVALUE AS L2
      WHERE
        L1.DOC_ID = did1 AND L2.DOC_ID = did2 AND
        L1.PATH_ID = L2.PATH_ID AND
        L1.LEAFVALUE = L2.LEAFVALUE ) AS T
   GROUP BY T.LEVEL, T.SO1, T.SO2 ) AS D,
  (SELECT L1.LEVEL, L1.SIBLINGORDER,
          COUNT(L1.LEAFORDER) AS TOTAL
   FROM LEAFVALUE AS L1 WHERE L1.DOC_ID = did1
   GROUP BY L1.LEVEL, L1.SIBLINGORDER) AS V1,
  (SELECT L1.LEVEL, L1.SIBLINGORDER,
          COUNT(L1.LEAFORDER) AS TOTAL
   FROM LEAFVALUE AS L1 WHERE L1.DOC_ID = did2
   GROUP BY L1.LEVEL, L1.SIBLINGORDER) AS V2
WHERE
   V1.LEVEL = D.LEVEL AND V2.LEVEL = D.LEVEL AND
   V1.SIBLINGORDER = D.SO1 AND V2.SIBLINGORDER = D.SO2
```

(b) Finding Matching SiblingOrder

```
SELECT
   A1.NODELEVEL,
   A1.MINSIBORDER AS MINSO1,
   A1.MAXSIBORDER AS MAXSO1,
   A2.MINSIBORDER AS MINSO2,
   A2.MAXSIBORDER AS MAXSO2,
   SUM(T.COUNTER) AS COUNTER,
   SUM(T.TOTAL) AS TOTAL
FROM
   ANCESTORINFO AS A1,
   ANCESTORINFO AS A2, TEMPSO AS T
WHERE
   A1.DOC_ID = did1 AND
   A2.DOC_ID = did2 AND
   T.SO1 BETWEEN A1.MINSIBORDER AND
      A1.MAXSIBORDER AND
   T.SO2 BETWEEN A2.MINSIBORDER AND
      A2.MAXSIBORDER AND
   A1.NODELEVEL = A2.NODELEVEL AND
   A1.NODENAME = A2.NODENAME AND
   A1.NODELEVEL = level
GROUP BY
  A1.NODELEVEL, A1.MINSIBORDER,
  A1.MAXSIBORDER, A2.MINSIBORDER,
  A2.MAXSIBORDER
```

(c) Finding Possible Matching Internal Node

Fig. 3. Algorithm *findBestMatchingSubtree* and SQL Queries.

rooted at nodes $i_1 \in I(T_1)$ and $i_2 \in I(T_2)$ respectively. Let name(i) and level(i) be the node name and node level of an internal node i respectively. S_1 and S_2 are the ***possible matching subtrees*** *if the following conditions are satisfied:* 1) *name(i_1) = name(i_2),* 2) *level(i_1) = level(i_2), and* 3) *$\exists P\ \exists Q$ such that $P \Leftrightarrow Q$ where $P \in S_1$ and $Q \in S_2$.*

We only consider matching subtrees in the same level for the same reason as in [10]. Next, we determine the *best matching subtrees* from a set of possible matching subtrees. Consequently, we have to measure how similar two possible matching subtrees are. Formally, *similarity score* can be defined as follows.

Definition 4. [**Similarity Score**] *The similarity score $\Re$ of two subtrees t_1 and t_2 as follows: $\Re(t_1, t_2) = \frac{2|t_1 \cap t_2|}{|t_1 \cup t_2|}$ where $|t_1 \cup t_2|$ is the total number of nodes of subtrees t_1 and t_2, and $|t_1 \cap t_2|$ is number of matching nodes.*

The similarity score will be between 0 and 1. Based on the similarity score, we are able to classify the matching subtree into three types: 1)**Isomorphic Subtrees** ($\Re(t_1, t_2) = 1$). We say two subtrees are isomorphic if they are identical except for the orders among siblings. 2)**Unmatching Subtrees** ($\Re(t_1, t_2) = 0$). We say two subtrees are unmatching if they are totally different. 3)**Matching Subtrees** ($0 < \Re(t_1, t_2) < 1$). The matching subtrees have some parts in the trees that are corresponded each other.

After we are able to determine how similar the possible matching subtrees are, the best matching subtrees can be determined. The formal definition of the best matching subtrees is as follows.

Definition 5. [**Best Matching Subtrees**] *Let $t \in T_1$ be a subtree in T_1 and $P \subseteq T_2$ be a set of subtrees in T_2. Also t and $t_i \in P$ are possible matching subtrees $\forall\ 0 < i \leq |P|$. Then t and t_i are the* ***best matching subtrees*** *(denoted by $t \backsimeq t_i$) iff $(\Re(t, t_i) > \Re(t, t_j))\ \forall\ 0 < j \leq |P|$ and $i \neq j$.*

The algorithm for determining the best matching subtrees is depicted in Figure 3(a). Given two XML trees T_1 and T_2 shredded in a relational database as shown in Figure 2 and the similarity score threshold (say θ=0.25), the *findBestMatchingSubtree* algorithm starts finding the matching best subtrees by checking the root nodes of T_1 and T_2 (lines 1-3, Figure 3(a)). If they have different names,

TempSO (Level, SO1, SO2, Counter, Total)

Matching (DID1, DID2, MinSO1, MaxSO1, MinSO2, MaxSO2, Level, Counter, Total, Score)

(a) Attributes of The TempSO and Matching Tables

Level	SO1	SO2	Counter	Total
3	1	1	2	7
3	1	3	4	7
3	2	2	4	7

(b) TempSO Table

ID1	ID2	DID1	DID2	Level	Min SO1	Max SO1	Min SO2	Max SO2	Counter	Total	Score
1	1	1	3	1	1	3	1	3	8	14	0.5714
7	106	2	2	2	2	2	2	2	4	7	0.5714
2	102	3	3	2	1	1	1	1	2	7	0.2857
2	111	3	3	2	1	1	3	3	4	7	0.5714

(c) Matching Table

```
DEL_INT (DID1, DID2, NAME, LEVEL, MINSO, MAXSO)
INS_INT (DID1, DID2, NAME, LEVEL, MINSO, MAXSO)
DEL_LEAF (DID1, DID2, LEVEL, SIBLINGORDER, PATH_ID, VALUE)
INS_LEAF (DID1, DID2, LEVEL, SIBLINGORDER, PATH_ID, VALUE)
UPD_LEAF (DID1, DID2, LEVEL, SO1, SO2, PATH_ID, VALUE1, VALUE2)
```

(d) Tables and Attributes

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

(e) Description of Attributes

Fig. 4. The `TempSO` and `Matching` Tables, and Table Description.

then both XML documents are considered as different. Consequently, the delta only consists of a deletion of T_1 and an insertion of T_2. Otherwise, the algorithm finds the *matching sibling orders* (line 4, Figure 3(a)). The SQL query for retrieving the matching sibling order is depicted in Figure 3(b). The results are stored in the `TempSO` table (Figure 4(b)) whose attributes are depicted in Figure 4(a).

Next, the *findBestMatchingSubtree* algorithm determines the deepest level *maxLevel* of the root nodes of subtrees in T_1 and T_2 (line 5, Figure 3(a)). For each level *curLevel* starting from level *maxLevel* to the level of the root nodes of the trees (level=1), the algorithm starts by finding the best matching subtrees (lines 6-10, Figure 3(a)). First, the algorithm finds the *possible matching internal nodes* (line 7, Figure 3(a)). The SQL query shown in Figure 3(c) is used to retrieve the *possible matching internal nodes*. We store the results in the `Matching` table whose attributes are depicted in Figure 4(a). The `Matching` table of T_1 and T_2 is depicted in Figure 4(c).

The next step is to maximize the similarity scores of the possible matching internal nodes at level *curLevel* at which the possible matching subtrees are rooted (line 8, Figure 3(a)) since we may have some subtrees and sibling orders at (*curLevel*+1) in T_1 that can be matched to more than one subtrees and sibling orders in T_2 respectively, and vice versa. The *maximizeSimilarityScore* algorithm is similar to the Smith-Waterman algorithm [9] for sequence alignments. Due to the space constraints, we do not present the *maximizeSimilarityScore* algorithm here. It can be found in [6]. For instance, the score of possible matching subtrees rooted at nodes 1 and 101 at level 1 is maximized if $t_2 \Bumpeq t_{111}$ and $t_7 \Bumpeq t_{106}$. The The corresponding tuple of the possible matching subtrees which are not used in maximizing the score are deleted (highlighted row, Figure 3(c)).

4 Phase 2: Detecting The Changes

In the second phase, first, we detect the inserted and deleted internal nodes. Then we find the inserted and deleted leaf nodes. Finally, we detect the updated leaf nodes from the inserted and deleted leaf nodes as they can be decomposed into pairs of deleted and inserted leaf nodes. The formal definitions of types of changes can be found in [6].

Insertion of Internal Nodes. Intuitively, the inserted internal nodes are the internal nodes that are in the new version, but not in the old version. Hence, they must be not the root nodes of the best matching subtrees as they are in both versions. The SQL query depicted in Figure 5(a) (*did1* and *did2* refer to the first and second versions of the document respectively) detects the set of newly inserted internal nodes. Consider the example in Figure 1. We notice that

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

(a) Insertion of Internal Nodes

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

(b) Insertion of Leaf Nodes (1)

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

(c) Insertion of Leaf Nodes (2)

```
1   SELECT T.*
2   FROM MATCHING AS M,
3     (SELECT
4        D.DID1, D.DID2, D.LEVEL,
5        D.SIBLINGORDER AS SO1,
6        I.SIBLINGORDER AS SO2,
7        D.PATH_ID, D.VALUE AS VALUE1,
8        I.VALUE AS VALUE2
9      FROM DEL_LEAF AS D, INS_LEAF AS I
10     WHERE
11       D.DID1 = did1 AND
12       D.DID2 = did2 AND
13       I.DID1 = did1 AND
14       I.DID2 = did2 AND
15       D.PATH_ID = I.PATH_ID AND
16       D.VALUE != I.VALUE AND
17       D.LEVEL = I.LEVEL ) AS T
18  WHERE
19    M.DID1 = did1 AND
20    M.DID2 = did2 AND
21    T.SO1 BETWEEN M.MINSO1 AND M.MAXSO1 AND
22    T.SO2 BETWEEN M.MINSO2 AND M.MAXSO2 AND
23    T.LEVEL = M.LEVEL+1
```

(d) Updated Leaf Nodes

Fig. 5. SQL Queries for Detecting the Changes.

the subtree rooted at node 102 in T_2 is inserted. The inserted internal nodes are retrieved by the SQL query depicted in Figure 5(a) and are stored in the `INS_INT` table as shown in Figure 6(a).

Deletion of Internal Nodes. We can use the same intuition to find the deleted internal nodes that are in T_1, but not in T_2. The deleted internal nodes can be detected by slightly modifying the SQL query depicted in Figure 5(a). We replace the "*did2*" in line 6 with "*did1*". The "`MINSO2`" and "`MAXSO2`" in line 8 are replaced by "`MINSO1`" and "`MAXSO1`" respectively. In the example shown in Figure 1, we observe that the subtree rooted at node 11 in T_1 is deleted. The deleted internal nodes are retrieved by the SQL query depicted in Figure 5(a) (after some modification) and are stored in the `DEL_INT` table as shown in Figure 6(b).

Insertion of Leaf Nodes. The *new* leaf nodes are only available in the second version of an XML tree. These new nodes should be either in the *best matching subtrees* or in the newly *inserted subtrees*. Consider the Figure 1. The leaf nodes 103, 104, and 105 belong to the newly *inserted subtree* rooted at node 102. The leaf node 109 is also inserted in the new version but it is contained in the best matching subtree rooted at node 106. Note that this subtree is not newly inserted one. We use two SQL queries to detect the two types of inserted leaf nodes as depicted in Figures 5(b) and (c). The SQL query shown in Figure 5(b) is used to detect the inserted leaf nodes that are in the newly inserted subtrees. The inserted leaf nodes that are in the matching subtrees are detected by using the SQL query shown in Figure 5(c). The result of the queries is stored in the `INS_LEAF` table as shown in Figure 6(c). Note that the highlighted tuples in Figure 6(c) are actually updated leaf nodes. However, they are detected as inserted nodes.

Deletion of Leaf Nodes. The *deleted* leaf nodes are only available in the first version of an XML tree. These deleted nodes should also be either in the *best matching subtrees* or in the *deleted subtrees*. Consider the Figure 1. The leaf nodes 12, 13, 14, and 15 belong to the *deleted subtree* rooted at node 11. The leaf node 5 is also deleted but it is contained in the best matching subtree rooted at node 2. We also use two SQL queries for detecting these two types of deleted leaf nodes. These SQL queries are generated by slightly modifying the queries in the Figures 5(b) and (c). We replace "`INS_INT`" in line 5 in Figure 5(b) with "`DEL_INT`". We also replace the "*did2*" in line 7 in Figure 5(b) and in lines 8 and

DID1	DID2	NAME	LEVEL	MINSO	MAXSO
1	2	product	2	1	1

(a) Inserted Internal Nodes

DID1	DID2	NAME	LEVEL	MINSO	MAXSO
1	2	product	2	1	1

(b) Deleted Internal Nodes

DID1	DID2	LEVEL	SIBLING ORDER	PATH_ ID	VALUE
1	2	3	1	1	Modem
1	2	3	1	2	56K Internal
1	2	3	1	4	US$ 42
1	2	3	2	2	APPLE iPod
1	2	3	2	3	15GB MP3 Player
1	2	3	3	2	V-GEN 256MB PC2700

(c) Inserted Leaf Nodes

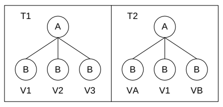
DID1	DID2	LEVEL	SIBLING ORDER	PATH_ ID	VALUE
1	2	3	1	2	V-GEN 2700/256
1	2	3	1	3	256MB PC-2700
1	2	3	2	2	APPLE iPod 15GB
1	2	3	3	1	Storage
1	2	3	3	2	USB Mini Cruzer
1	2	3	3	3	128MB USB 2.0
1	2	3	3	4	US$ 32

(d) Deleted Leaf Nodes

DID1	DID2	LEVEL	SO1	SO2	PATH_ID	VALUE1	VALUE2
1	2	3	2	2	2	APPLE iPod 15GB	APPLE iPod
1	2	3	1	3	2	V-GEN 2700/256	V-GEN 256MB PC2700

(e) Updated Leaf Nodes

Fig. 6. Detected Delta.



(a) Example

DID1	DID2	LEVEL	SO1	SO2	PATH_ ID	VALUE 1	VALUE 2
..	..	..	..	..	..	V2	VA
..	..	..	..	..	..	V2	VB
..	..	..	..	..	..	V3	VA
..	..	..	..	..	..	V3	VB

(b) Updated Leaf Nodes

```
Input: Table UPD_LEAF, doc_id1, doc_id2
Output: Corrected Table UPD_LEAF

Algorithm updateCorrector {
  while (result R of query Q1 is not empty){
    correctUpdateTable(R);
  }
  while (result R of query Q2 is not empty){
    correctUpdateTable(R);
  }
}
```

(c) Algorithm *updateCorrector*

```
SELECT
   U.SO1, U.SO2, U.PATH_ID,
   U.VALUE1, U.VALUE2
FROM UPD_LEAF AS U,
  (SELECT DID1, DID2, SO1,
          SO2, PATH_ID,
          VALUE1, COUNT(VALUE1)
   FROM UPD_LEAF
   WHERE DID1 = doc_id1 AND
         DID2 = doc_id2
   GROUP BY DID1, DID2, SO1, SO2,
            PATH_ID, VALUE1
   HAVING COUNT(VALUE1)>1) AS T
WHERE
   U.DID1 = doc_id1 AND
   U.DID2 = doc_id2 AND
   U.SO1 = T.SO1 AND
   U.SO2 = T.SO2 AND
   U.PATH_ID = T.PATH_ID AND
   U.VALUE1 = T.VALUE1
FETCH FIRST 1 ROWS ONLY
OPTIMIZE FOR 1 ROWS
```

(d) SQL Query (1)

```
DELETE FROM UPD_LEAF
WHERE
  DID1 = doc_id1 AND  DID2 = doc_id2 AND
  SO1 = R.SO1 AND SO2 = R.SO2 AND
  PATH_ID = R.PATH_ID AND
  ((VALUE1 = R.VALUE1 AND VALUE2 != R.VALUE2) OR
   (VALUE1 != R.VALUE1 AND VALUE2 = R.VALUE2))
```

(e) SQL Query (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

(f) Sigmod Dataset

Fig. 7. Example Uncomplete Results of Update Query and Datasets.

20 in Figure 5(c) with "*did1*". The "*did1*" in line 16 in Figure 5(c) is replaced by "*did2*". We also replace "`MINSO2`" and "`MAXSO2`" in lines 10 and 21 in Figure 5(c) with "`MINSO1`" and "`MAXSO1`" respectively. The "`MINSO1`" and "`MAXSO1`" in line 18 in Figure 5(c) are replaced by "`MINSO2`" and "`MAXSO2`" respectively. Figure 6(d) depicts the result of the queries which is stored in the `DEL_LEAF` table. Note that the highlighted rows are actually updated leaf nodes which are detected as deleted leaf nodes.

Content Update of Leaf Nodes. Intuitively, an updated node is available in the first and second versions, but its value is different. As the updated leaf nodes are detected as pairs of deleted and inserted leaf nodes, we are able to find the updated leaf nodes from two sets of leaf nodes: the *inserted leaf nodes* and the *deleted leaf nodes* respectively. In addition, we also need the information of the best matching subtrees in order to guarantee the updated leaf nodes are in the best matching subtrees. Note that we only consider the update of the content of the leaf nodes. Similar to [10], the modification of the name of an internal node is detected as a pair of deletion and insertion. The SQL query for detecting the updated leaf nodes is depicted in Figure 5(d) and the results are in the `UPD_LEAF` table. The updated leaf nodes of the example in Figure 1 are shown in Figure 6(e) (the `UPD_LEAF` table). We observe that the result of this SQL query may not be correct result in some cases. Let us elaborate further. Suppose we have two trees as depicted in Figure 7(a). The result of the SQL query depicted in Figure 5(f) is shown in Figure 7(b) (partial view only). We notice that nodes `B` with values "V2" and "V3" are detected as updated leaf nodes twice. This is because the sub query in lines 3-17 in Figure 5(d) only finds the leaf nodes which have the same paths, but different values. We use the *updateCorrector* algorithm that is depicted in Figure 7(c) to correct the result by finding the *incorrect tuples*. A tuple t is an *incorrect tuple* if one and only one of the following conditions is satisfied: 1) the `VALUE1` of tuple t is equal to `VALUE1` of tuple R, 2) the `VALUE2` of

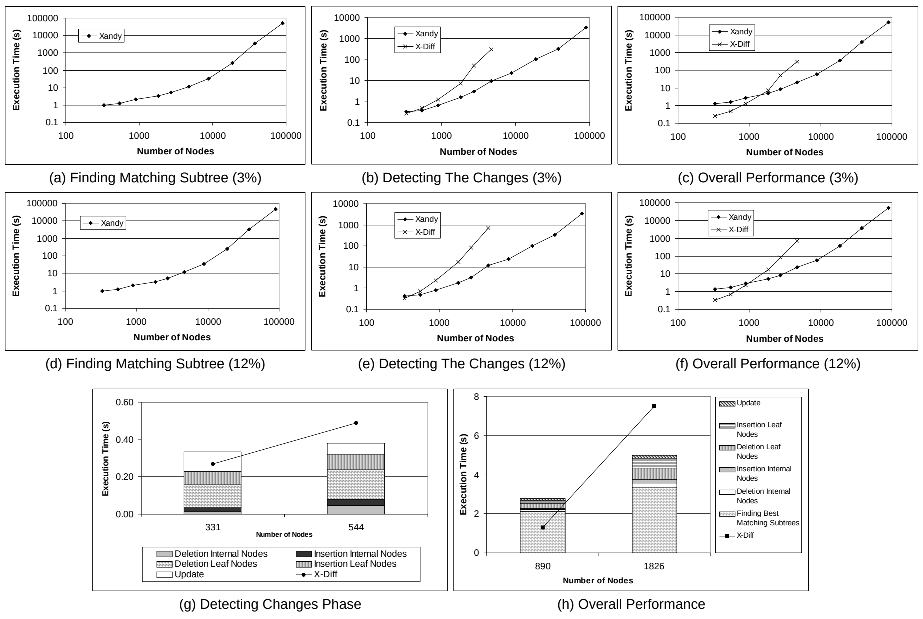


Fig. 8. Execution Time vs Number of Nodes (Logarithmic Scale)

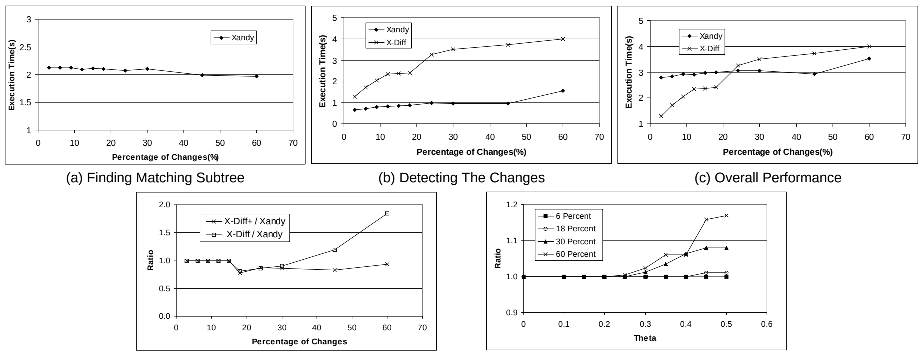
tuple t is equal to `VALUE2` of tuple R. The algorithm iteratively issues the SQL queries depicted in Figures 7(d) and (e) until no *incorrect tuple* is found.

5 Performance Study

We have implemented XANDY 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 (Figure 7(f)). 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. The experimental results that support this decision are available in [6]. 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 compared the performance of XANDY to the Java version of X-Diff [2].

Execution Time vs Number of Nodes. In this set of experiments, we study the performance of XANDY for different number of nodes. The percentages of changes are set to "3%" and "12%" and the threshold θ is set to "0.0" which shall give us the upper bound of the execution time. Figures 8(a) and (d) show the performance of the first phase (*Finding Best Matching*) when we set the percentages of changes to 3% and 12% respectively. For XML documents that have less than 5000 nodes, the execution time of the first phase is less than 12

[2] downloaded from www.cs.wisc.edu/~yuanwang/xdiff.html



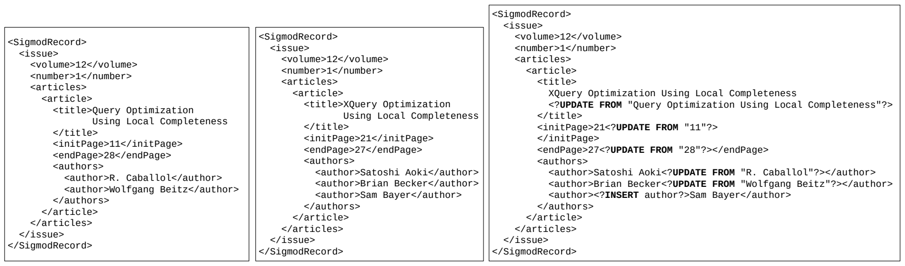
(a) Finding Matching Subtree (b) Detecting The Changes (c) Overall Performance

(d) Result Quality (1) (e) Result Quality (2)

Fig. 9. Execution Time vs Percentage of Changes and Result Quality

seconds. Figures 8(b) and (e) show the performance of the second phase (*Detecting the Changes*) compared to X-Diff when we set the percentage of changes to 3% and 12% respectively. We observe that XANDY performs better than X-Diff except for the smallest data set. Figure 8(g) depicts the performance comparison between X-Diff and second phase of XANDY for "Sigmod-01" and "Sigmod-02". We observe that most of the execution time of the second phase is taken by finding the updated leaf nodes, detecting the inserted leaf nodes, and detecting the deleted leaf nodes. Even then, it is faster than X-Diff (for "Sigmod-02" dataset). In this experiment, X-Diff is unable to detect the changes on the XML documents that have number of nodes over 5000 nodes due to lack of the main memory. Figures 8(c) and (f) show the overall performance of XANDY compared to X-Diff when we set the percentages of changes to 3% and 12% respectively. We notice that the difference of execution time between XANDY and X-Diff reduces as the number of nodes increases. Finally, XANDY becomes faster than X-Diff after the number of nodes is greater than 1000 nodes. This is because the query engine of the relational database is still able to process the data efficiently as the increment of the size of data is not significant. Figure 8(h) depicts the comparison between X-Diff and of XANDY for the third and fourth datasets. We observe that the first phase takes up to 70% of the overall execution time in average. XANDY is able to detect the changes on XML documents with over 89,000 nodes. From these experiments, we conclude that XANDY has better scalability than X-Diff. For small datasets, XANDY has comparable performance compared to X-Diff. XANDY has better performance than X-Diff for the large datasets.

Execution Time vs Percentage of Changes. In this set of experiments, we use the dataset "Sigmod-03" and the threshold θ is set to "0.0". We vary the percentages of changes from "3%" to "60%". Figure 9(a) depicts the execution time of the first phase in XANDY. We observe that the percentage of changes influence the execution time for finding the best matching subtrees. This is because there will be more number of matching sibling orders when the documents are changed slightly. On the other hand, when the documents are changed significantly, we will have lesser number of matching sibling orders. Figure 9(b) depicts the execution time of the second phase in XANDY. We observe that XANDY outperforms



```
<SigmodRecord>
  <issue>
    <volume>12</volume>
    <number>1</number>
    <articles>
      <article>
        <title>Query Optimization
               Using Local Completeness
        </title>
        <initPage>11</initPage>
        <endPage>28</endPage>
        <authors>
          <author>R. Caballol</author>
          <author>Wolfgang Beitz</author>
        </authors>
      </article>
    </articles>
  </issue>
</SigmodRecord>
```

(a) First Version

```
<SigmodRecord>
  <issue>
    <volume>12</volume>
    <number>1</number>
    <articles>
      <article>
        <title>XQuery Optimization
               Using Local Completeness
        </title>
        <initPage>21</initPage>
        <endPage>27</endPage>
        <authors>
          <author>Satoshi Aoki</author>
          <author>Brian Becker</author>
          <author>Sam Bayer</author>
        </authors>
      </article>
    </articles>
  </issue>
</SigmodRecord>
```

(b) Second Version

```
<SigmodRecord>
  <issue>
    <volume>12</volume>
    <number>1</number>
    <articles>
      <article>
        <title>
          XQuery Optimization Using Local Completeness
          <?UPDATE FROM "Query Optimization Using Local Completeness"?>
        </title>
        <initPage>21<?UPDATE FROM "11"?>
        </initPage>
        <endPage>27<?UPDATE FROM "28"?></endPage>
        <authors>
          <author>Satoshi Aoki<?UPDATE FROM "R. Caballol"?></author>
          <author>Brian Becker<?UPDATE FROM "Wolfgang Beitz"?></author>
          <author><?INSERT author?>Sam Bayer</author>
        </authors>
      </article>
    </articles>
  </issue>
</SigmodRecord>
```

(c) Delta of X-Diff+

Fig. 10. Example.

the X-Diff. We also notice that the execution times of XANDY and X-Diff are affected by the percentage changes. Figure 9(c) shows the overall performance. X-Diff is faster than XANDY for the percentage of changes less than around 20%. As the percentage of changes is larger than 20%, XANDY becomes faster than X-Diff. This is because the time for finding the best matching subtrees is reduced as the percentage of changes is increased.

Result Quality. In the first experiment, we examine the effect of the percentage of changes on the *result quality* by using "Sigmod-03" as the data set. A series of new versions are generated by varying the percentage of the changes. XANDY, X-Diff, and X-Diff+ [3] were run to detect the changes on these XML documents. The number of nodes involved in the deltas is counted for each approach. We compare the number of nodes in the deltas detected by XANDY to the one detected by X-Diff, and X-Diff+. The ratios are plotted in Figure 9(d). We observed that XANDY detects the same deltas as X-Diff+ until the percentage of the changes reaches 15%. The quality ratios of X-Diff+ and XANDY are smaller than 1 when the percentage of the changes is larger than 15%. This happens because X-Diff+ detects a deletion and insertion of subtrees as a set of update operations. For example, we have two versions of an XML document as depicted in Figures 10(a) and (b). Figure 10(c) depicts the delta detected by X-Diff+. XANDY detects as a deletion of an article and an insertion of an article. We notice that the quality ratios of X-Diff and XANDY are larger than 1 when the percentage of the changes are larger than 30%. This is because X-Diff does not calculate the minimum editing distance. Consequently, X-Diff may detect as a deletion of a subtree if it is changed significantly. Note that this does not happen on X-Diff+ as it calculates the minimum editing distance.

In the second experiment, we study the effect of the similarity threshold θ in our approach on the *result quality* by using "Sigmod-04" data set. Then a series of new versions are generated by setting the percentages of the changes to 6%, 18%, 30%, and 60%. For each percentage of changes, XANDY was run by varying threshold θ. The number of nodes involved in the deltas is counted for each threshold θ. We compare the number of nodes in the deltas detected by XANDY with $\theta = 0.0$ to the one detected by XANDY with $0.10 \leq \theta \leq 0.50$. The ratios are plotted in Figure 9(e). We observe that the threshold θ may not affect the result quality if the documents are changed slightly. On the other hand, the result

[3] We activate the option "-o" of X-Diff so it calculates the minimum editing distance in finding the matchings.

quality is affected by the threshold θ if the documents are changed significantly. When the percentage of changes is set to 60%, the result quality becomes worse as the threshold $\theta \geq 0.25$. We conclude that the result quality of the deltas detected by XANDY is influenced by the percentage of changes, the distribution of the changes, and the threshold θ. The distribution of the changes influences the result quality in the following way. Suppose we have a subtree t_1 in which the changes are concentrated. t_2 is the matching subtree of t_1. The similarity score $\Re(t_1,t_2)$ will be reduced as t_1 and t_2 have less common nodes. Consequently, t_1 and t_2 may be considered as unmatching subtrees if $\Re(t_1,t_2) < \theta$.

6 Conclusions

The relational approach for unordered XML change detection system in this paper is motivated by the scalability problem of existing main memory-based approaches. We have shown that the relational approach is able to handle XML documents that are much larger than the ones detected by using main-memory approaches. In summary, the number of nodes and the percentage of changes influence the execution time of all approaches. XANDY is able to detect the changes on XML documents with over 89,000 nodes, while X-Diff is only able to detect the changes the XML documents with up to 5,000 nodes. We also show that the execution of XANDY is faster than X-Diff for large data sets. This shows that the powerful query engine of the relational database can be utilized for the detecting the changes. The result quality of XANDY is comparable to the one of X-Diff. In XANDY, the result quality depends on the threshold θ, the percentage of changes, and the distribution of the changes.

References

1. YAN CHEN, S. MADRIA, S. S. BHOWMICK. DiffXML: Change Detection in XML Data. *DASFAA 2004*, Jeju Island, Korea, 2004.
2. CURBERA, D. A. EPSTEIN. Fast Difference and Update of XML Documents. *XTech'99,* San Jose, 1999.
3. G. COBENA, S. ABITEBOUL, A. MARIAN. Detecting Changes in XML Documents. *ICDE 2002*, San Jose, 2002.
4. H. JIANG, H. LU, W. WANG, J. XU YU. Path Materialization Revisited: An Efficient Storage Model for XML Data. *Australasian Database Conference*, Melbourne, Australia, 2002.
5. ERWIN LEONARDI, S. S. BHOWMICK, S. MADRIA. Detecting Content Changes on Ordered XML Documents Using Relational Databases. *DEXA 2004*, Zaragoza, Spain, 2004.
6. ERWIN LEONARDI, S. S. BHOWMICK. XANDY: Detecting Changes on Large Unordered XML Documents Using Relational Database. *Technical Report, Center for Advanced Information System, Nanyang Technological University*, Singapore, 2004. http://www.cais.ntu.edu.sg/∼erwin/docs/
7. S. PRAKASH, S. S. BHOWMICK, S. MARDIA. SUCXENT: An Efficient Path-based Approach to Store and Query XML Documents. *DEXA 2004*, Spain, 2004.
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. T. F. SMITH AND M. S. WATERMAN Identification of common molecular subsequences. *Journal Molecular Biology* 147:195-197, 1981.
10. Y. WANG, D. J. DEWITT, J. CAI. X-Diff: An Effective Change Detection Algorithm for XML Documents. *ICDE 2003*, Bangalore, 2003.