Detecting Content Changes on Ordered XML Documents Using Relational Databases

Erwin Leonardi[1] Sourav S. Bhowmick[1] T.S. Dharma[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 focused on detecting changes to text file using ordered and unordered tree model. These approaches are not suitable for detecting changes to large XML document 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 *content* changes of *ordered* large XML data. First, we store XML documents in RDBMS. Then, we detect the changes by using a set of SQL queries. Experimental results show that our approach has better scalability, better performance, and comparable result quality compared to the state-of-the-art approaches.

1 Introduction

Over the next few years XML is likely to replace HTML as the standard format for publishing and transporting documents over the Web. Since online information changes frequently, being able to quickly detect the changes in XML documents (hereafter called *XML deltas* or *XDeltas*) is an important problem. Such change detection tool is important to incremental query evaluation, trigger condition evaluation, search engine, data mining applications, and mobile applications [1]. In this paper, we focus on detecting *content* changes on the *ordered* XML documents.

Let us illustrate changes to ordered XML with an example. Suppose we have two versions of an XML document that is used for storing the contents of a book. These XML documents are represented as trees as depicted in Figure 1. These XML documents are classified into ordered XML since the content of a book must be ordered. The highlighted ellipses in Figure 1 indicate the different types of changes. The "XML Syntax" chapter is moved to be the second chapter of the book because of insertion of "XML Introduction" chapter.

The changes on ordered XML documents can be classified into two types: *changes occur in the internal element* and *changes occur in the leaf element.* An *internal element* is the element which does not contain textual data and is not a leaf element. For example, nodes 5 and 6 in Figure 1(a) are the internal elements. The changes occur in the internal elements (which are called as *structural changes*) modify the structure and do not change textual data content. The types

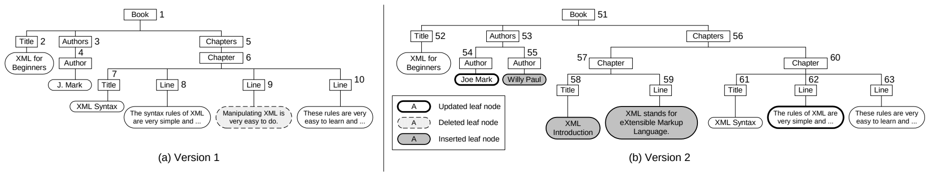


Fig. 1. Tree representation of Ordered XML Documents.

of changes which are classified as *structural changes* are as following: *insertion of internal element*, *deletion of internal element*, and *internal element movement*. For instance, node 57 in Figure 1(b) is the new inserted element. Node 60 is moved from being the first child of node 56 to be the second child of node 56. A *leaf element* is the element/attribute which contains textual data. For example, node 52 is a leaf element which has name "Title" and textual content "XML for Beginners". The changes in the leaf elements (which are called as *content changes*) modify textual data content. There are four types of *content changes* as following: *leaf element insertion*, *leaf element deletion*, *content update of leaf element*, and *leaf element movement*. A leaf element "Line" which has value "Manipulating XML is very easy to do." is a new inserted leaf element. In this paper, we present a technique for detecting *content* changes (changes to the leaf element) on the *ordered* XML documents using relational database.

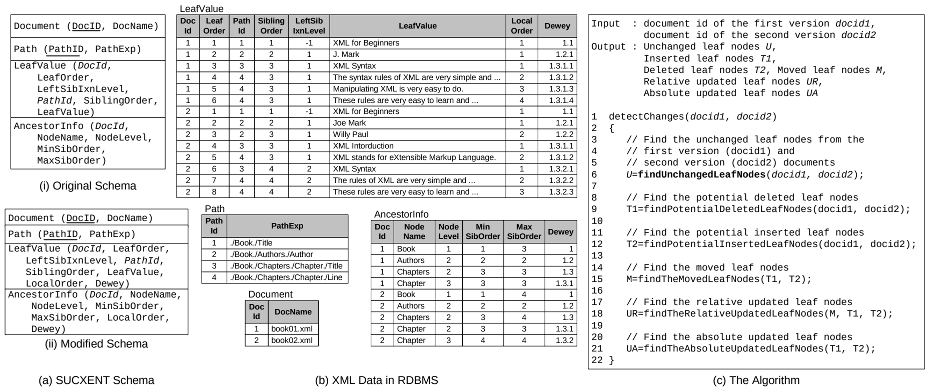
Some previous works [1–3] have been proposed to solve the problem of detecting changes on XML documents. Cobena et al. [2] proposed an algorithm, called XyDiff, for detecting changes on ordered XML documents. The changes are detected by using the signature and weight of nodes. XMLTreeDiff [3] is also proposed for solving the problem of detecting changes for ordered XML documents by using DOMHash. In [1], the authors presented X-Diff, an algorithm for detecting the changes on unordered XML documents. The algorithm assigns XHash value for each node in the trees. The XHash value of a node is calculated from the XHash values of its descendants. The algorithm tries to find a minimum-cost matching between two documents before generating the edit script in order to find the minimum edit script.

Our approach is different from the previous approaches in the following ways. First, our approach focuses on the *content change*, while the state-of-art approaches detect both *structural changes* and *content changes*. Second, we detect the changes on XML documents by using relational database. The state-of-art approaches detect the changes on XML documents stored in main memory after they are parsed. Our approaches give the opportunities to have more scalable change detection system since we depend on the secondary storage rather than the main memory. We store the XML documents in a relational database. Then, we detect the changes by using a set of SQL queries.

2 Storing XML Data

There are two approaches for storing XML documents in relational database: the *structure-mapping* and the *model-mapping* approaches [5, 7]. In this paper, we have decided to adopt the model mapping approach.

We now discuss how XML documents are stored in relational database by using SUCXENT (**S**chema **U**n**C**oncious **X**ML **EN**abled sys**T**em) schema [7].



```
Document (DocID, DocName)

Path (PathID, PathExp)

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

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

(i) Original Schema

```
Document (DocID, DocName)

Path (PathID, PathExp)

LeafValue (DocId, LeafOrder,
   LeftSibIxnLevel, PathId,
   SiblingOrder, LeafValue,
   LocalOrder, Dewey)

AncestorInfo (DocId, NodeName,
    NodeLevel, MinSibOrder,
    MaxSibOrder, LocalOrder,
    Dewey)
```

(ii) Modified Schema

(a) SUCXENT Schema

LeafValue

Doc Id	Leaf Order	Path Id	Sibling Order	LeftSib IxnLevel	LeafValue	Local Order	Dewey
1	1	1	1	-1	XML for Beginners	1	1.1
1	2	2	2	1	J. Mark	1	1.2.1
1	3	3	3	1	XML Syntax	1	1.3.1.1
1	4	4	3	1	The syntax rules of XML are very simple and ...	2	1.3.1.2
1	5	4	3	1	Manipulating XML is very easy to do.	3	1.3.1.3
1	6	4	3	1	These rules are very easy to learn and ...	4	1.3.1.4
2	1	1	1	-1	XML for Beginners	1	1.1
2	2	2	2	1	Joe Mark	1	1.2.1
2	3	2	3	1	Willy Paul	2	1.2.2
2	4	3	3	1	XML Intorduction	1	1.3.1.1
2	5	4	3	1	XML stands for eXtensible Markup Language.	2	1.3.1.2
2	6	3	4	2	XML Syntax	1	1.3.2.1
2	7	4	4	2	The rules of XML are very simple and ...	2	1.3.2.2
2	8	4	4	2	These rules are very easy to learn and ...	3	1.3.2.3

Path

Path Id	PathExp
1	./Book./Title
2	./Book./Authors./Author
3	./Book./Chapters./Chapter./Title
4	./Book./Chapters./Chapter./Line

Document

Doc Id	DocName
1	book01.xml
2	book02.xml

AncestorInfo

Doc Id	Node Name	Node Level	Min SibOrder	Max SibOrder	Dewey
1	Book	1	1	3	1
1	Authors	2	2	2	1.2
1	Chapters	2	3	3	1.3
1	Chapter	3	3	3	1.3.1
2	Book	1	1	4	1
2	Authors	2	2	2	1.2
2	Chapters	2	3	4	1.3
2	Chapter	2	3	3	1.3.1
2	Chapter	3	4	4	1.3.2

(b) XML Data in RDBMS

```
Input  : document id of the first version docid1,
         document id of the second version docid2
Output : Unchanged leaf nodes U,
         Inserted leaf nodes T1,
         Deleted leaf nodes T2, Moved leaf nodes M,
         Relative updated leaf nodes UR,
         Absolute updated leaf nodes UA

1  detectChanges(docid1, docid2)
2  {
3     // Find the unchanged leaf nodes from the
4     // first version (docid1) and
5     // second version (docid2) documents
6     U=findUnchangedLeafNodes(docid1, docid2);
7
8     // Find the potential deleted leaf nodes
9     T1=findPotentialDeletedLeafNodes(docid1, docid2);
10
11    // Find the potential inserted leaf nodes
12    T2=findPotentialInsertedLeafNodes(docid1, docid2);
13
14    // Find the moved leaf nodes
15    M=findTheMovedLeafNodes(T1, T2);
16
17    // Find the relative updated leaf nodes
18    UR=findTheRelativeUpdatedLeafNodes(M, T1, T2);
19
20    // Find the absolute updated leaf nodes
21    UA=findTheAbsoluteUpdatedLeafNodes(T1, T2);
22 }
```

(c) The Algorithm

Fig. 2. SUCXENT.

SUCXENT is a model-mapping, path-oriented approach. We choose SUCXENT because it outperforms significantly current state-of-the-art model mapping approaches like XParent [5] as far as storage size, insertion time, extraction time, and path expression queries are concerned [7]. SUCXENT schema is shown in Figure 2(a)(i). The details of the schema can be found in [7].

We modify the SUCXENT schema by adding the attribute *LocalOrder* in the `LeafValue` table to store the position of a leaf element among its siblings. We also need to know the ancestors' local orders of each leaf and internal elements. We adopt the Dewey Order Encoding [6]. Let us name this attribute *Dewey*. We add attribute *Dewey* in the `LeafValue` and `AncestorInfo` tables. The modified SUCXENT schema is depicted in Figure 2(a)(ii). Suppose we have two versions of an XML document that are represented as trees depicted in Figure 1. The XML documents stored in RDBMS by using SUCXENT schema are depicted in Figure 2(b).

3 Content Changes Detection

In this section, we discuss how to detect the content changes by using our approach. The algorithm of our approach is shown in Figure 2(c).

3.1 Unchanged Leaf Elements Detection Phase

In the first phase, we find the leaf elements that are not changed during the transition. The unchanged leaf elements must have the same paths (*PathID*), values (*LeafValue*), local orders (*LocalOrder*), and the ancestors' local orders (*Dewey*).

Given a set of tuples in the `LeafValue` table which correspond to leaf elements of two versions of an XML document, we are able to detect the unchanged leaf elements by using the SQL query shown in Figure 3(a). The SQL query is encapsulated in function `findUnchangedLeafNodes` (line 6) in Figure 2(c). Note that *docid1* and *docid2* are the document ids of the first and second versions respectively. Basically, the query tries to find the common leaf elements of the first and the second versions. We use "INTERSECT ALL" SQL operator because we want to preserve the duplicate tuples of two tables. The results of the queries are stores in the table, namely `UNCHANGED` table as depicted in Figures 4(a)(i).

```
//  Get the leaf elements in
//  the first document
SELECT
   PATHID, LEAFVALUE AS VALUE,
   LOCALORDER, DEWEY
FROM LEAFVALUE
WHERE DOCID = docid1
//  Find leaf elements occur in
//  both document
INTERSECT ALL
//  Get the leaf elements in the
//  second document
SELECT
   PATHID, LEAFVALUE AS VALUE,
   LOCALORDER, DEWEY
FROM LEAFVALUE
WHERE DOCID = docid2
// The result will be stored in
// table UNCHANGED
```

(a) Get Unchanged Leaf Elements

```
//  Get the leaf elements in the
//  document with document id = d1
SELECT
   PATHID, LEAFVALUE,
   LOCALORDER, DEWEY
FROM LEAFVALUE
WHERE DOCID = d1
//  Find leaf elements only occur
//  in document with id = d1
EXCEPT ALL
//  Get the leaf elements in the
//  document with document id = d2
SELECT
   PATHID, LEAFVALUE,
   LOCALORDER, DEWEY
FROM LEAFVALUE
WHERE DOCID = d2
```

(b) Get Leaf Elements Only in Document *d1*

```
SELECT
   T1.PATHID, T1.VALUE,
   T1.LOCALORDER,
   T2.LOCALORDER,
   T1.DEWEY, T2.DEWEY
FROM T1, T2
WHERE
   T1.PATHID = T2.PATHID AND
   T1.VALUE= T2.VALUE AND
   T1.DEWEY != T2.DEWEY AND
   T1.DEWEY NOT IN
     (SELECT DEWEY
      FROM UNCHANGED) AND
   T2.DEWEY NOT IN
     (SELECT DEWEY
      FROM UNCHANGED)
```

(c) Get Moved Leaf Elements

```
// A1 and A2 are the moved ancestors

SELECT
   T1.PATHID, T1.VALUE,
   T2.VALUE,
   T1.LOCALORDER, T1.DEWEY,
   T2.DEWEY
FROM T1, T2
WHERE
  T1.LOCALORDER=T2.LOCALORDER AND
  T1.PATHID=T2.PATHID AND
  T1.VALUE!=T2.VALUE AND
  T1.DEWEY LIKE 'A1.DEWEY%' AND
  T2.DEWEY LIKE 'A2.DEWEY%' AND
  (T1.DEWEY - A1.DEWEY)=
           (T2.DEWEY - A2.DEWEY)

// T1.DEWEY - A1.DEWEY means the rest
// part of a node's string after being
// cut A1.DEWEY from the begining
```

(d) Get Relative Updated Leaf Elements

Fig. 3. SQL Queries.

3.2 Potential Inserted and Deleted Leaf Elements Detection Phase

The objective of the second phase is to find the *potential* inserted and *potential* deleted leaf elements. This phase is useful to filter out data which are not used for detecting changes in the subsequent phases. We call them *potential* inserted and *potential* deleted leaf elements because they are not only the deleted and inserted leaf elements respectively, but also the updated leaf elements which are detected as pairs of deleted and inserted leaf elements respectively. For example, the corresponding tuples in the `T1` and `T2` tables of *potential* deleted and *potential* inserted leaf elements as depicted in Figures 4(a)(ii) and (iii) respectively. The first tuples in the `T1` and `T2` tables are the corresponding tuples of an updated leaf element which is detected as a pair of *potential* deleted and *potential* inserted leaf elements respectively.

Given a set of tuples in the `LeafValue` table which correspond to leaf elements of two versions of an XML document, we can detect the *potential* deleted and *potential* inserted leaf elements by using the SQL query as depicted in Figures 3(b). The SQL query is encapsulated in functions `findPossibleDeletedLeafNodes` (line 9) and `findPossibleInsertedLeafNodes` (line 12) in Figure 2(c). We use "EXCEPT ALL" SQL operator in order to preserve the duplicate tuples of two tables. If we want to find the *potential* deleted nodes, we set the value of *d1* and *d2* to the document ids of the first and second versions respectively. The value of *d1* and *d2* are set to the document ids of the second and first versions respectively if we want to get the *potential* inserted nodes. In the subsequent phases, we are going to detect the content changes of these XML documents by using the `UNCHANGED`, `T1` and `T2` tables.

3.3 Moved Leaf Elements Detection Phase

Next, we search for the moved leaf elements (if any) by using the results of the first and second phases. There are three types of moved leaf elements as follows. First, the leaf elements are moved because of the movement of their ancestors. Second, they are moved because of the insertions or deletions of their siblings. Third, the leaf elements are moved among their siblings.

Give two sets of tuples in the `T1` and `T2` tables, we execute a SQL query as depicted in Figure 3(c) in order to detect these moved leaf elements. The SQL query is encapsulated in function `findTheMovedLeafNodes` (line 15) in Figure 2(c). Basically, the query tries to find tuples in the `T1` and `T2` tables which have the same paths (*PathID*), and values (*Value*), but have different

Path ID	Value	Local Order	Dewey
1	XML for Beginner	1	1.1

(i) Table UNCHANGED

Path ID	Value	Local Order	Dewey	
2	J. Mark	1	1.2.1	
3	XML Syntax	1	1.3.1.1	
4	The syntax rules of XML are ...	2	1.3.1.2	@ #
4	Manipulating XML is very ...	3	1.3.1.3	
4	These rules are very easy ...	4	1.3.1.4	

(ii) Table T1

Path ID	Value	Local Order	Dewey	
2	Joe Mark	1	1.2.1	
2	Willy Paul	2	1.2.2	
3	XML Introduction	1	1.3.1.1	
4	XML stands for eXtensible ...	2	1.3.1.2	@
3	XML Syntax	1	1.3.2.1	
4	The rules of XML are ...	2	1.3.2.2	#
4	These rules are very easy ...	3	1.3.2.3	

(iii) Table T2

Path ID	Value	Local Order1	Local Order2	Dewey 1	Dewey 2
3	XML Syntax	1	1	1.3.1.1	1.3.2.1
4	These rules are very easy ...	4	3	1.3.1.4	1.3.2.3

(iv) Table Move

(a) Tables UNCHANGED, T1, T2, and Move

```
// A1 and A2 are the moved ancestors

SELECT
    T1.PATHID, T1.VALUE, T2.VALUE,
    T1.LOCALORDER, T1.DEWEY
FROM T1, T2
WHERE
    T1.LOCALORDER = T2.LOCALORDER AND
    T1.PATHID = T2.PATHID AND
    T1.VALUE != T2.VALUE AND
    T1.DEWEY = T2.DEWEY AND
    T1.DEWEY NOT LIKE 'A1.DEWEY%' AND
    T2.DEWEY NOT LIKE 'A2.DEWEY%';
```

(b) SQL Query: Get Absolute Updated Leaf Elements

Fig. 4. Tables and SQL Query.

PathID	Value1	Value2	LocalOrder	Dewey1	Dewey2
4	The syntax rules of XML are ...	The rules of XML are ...	1	1.1.2.1	1.2.2.1

(a) Relative Updated Leaf Elements (UR)

PathID	Value1	Value2	LocalOrder	Dewey
2	J. Mark	Joe Mark	1	1.2.1

(b) Absolute Updated Leaf Elements (UA)

Fig. 5. Updated Leaf Elements Detection.

Dewey value. If the local order of a leaf element is changed, the *Dewey* value of this node will also be changed. Similarly, the *Dewey* value of this node will be changed if the ancestors' local orders of a leaf element are changed. The results of this query are stored in the `Move` table which is depicted in Figure 4(a)(iv). Since these leaf elements have been detected as moved leaf elements, we have to delete the corresponding tuples of these leaf elements in the `T1` and `T2` tables. The highlighted rows in the `T1` and `T2` tables in Figures 4(a)(ii) and (iii) respectively are the deleted rows.

3.4 Relative Updated Leaf Elements Detection Phase

There are two types of updated leaf elements, namely, *absolute updated leaf elements* and *relative updated leaf elements*. If a leaf element which has the *same* absolute position in the first and second versions is updated, then we classify it as an *absolute updated leaf element*. We classify a leaf element as a *relative updated leaf element* if it is updated and has *different* absolute positions in the first and second versions. For example, the leaf element "Line" which has value "The syntax rules of XML are ..." is updated to "The rules of XML are ...". This leaf element has different absolute positions in the first and second versions. Detecting *absolute updated leaf elements* will be after detecting *relative updated leaf elements* in our approach in order to avoid misdetection.

We illustrate misdetection with an example. Suppose we have two sets of leaf elements which are stored as tuples in the `T1` and `T2` tables as depicted in Figure 4(a)(ii) and (iii) respectively after the highlighted rows are deleted. If we detect the absolute updated leaf elements first, the leaf element which is stored as tuple in the `T1` table with "1.3.1.2" as the value of its *Dewey* attribute (denoted by "@" at the right of the rows) will be detected as an updated leaf element from "The syntax rules of XML are ..." to "XML stands for eXtensible ..." since they have the same absolute position in both XML documents. The leaf element which has value "The syntax rules of XML are ..." should be updated to "The rules of XML are ..." (denoted by "♯" at the right of the rows). The leaf element which has value "XML stands for eXtensible ..." should be detected as an inserted leaf element.

We now discuss how we detect the relative updated leaf elements. The discussion on how we detect the absolute updated leaf elements will be presented in the later section. Before finding the relative updated leaf elements, we have to do some preprocessing on the *Dewey* values. The aim of this preprocessing is to find the ancestor nodes of the moved leaf elements which are moved. We need to find these moved ancestor nodes in order to know which subtrees in XML trees are moved. To find these moved ancestor nodes, we use the values of attributes *Dewey1* and *Dewey2* in the `Move` table. For example, the first tuple in the `Move` table has "1.3.**1**.1" and "1.3.**2**.1" in attributes *Dewey1* and *Dewey2* respectively. We notice that the parent of the corresponding leaf elements is moved to be the second child. The *Dewey* values of these moved ancestors are "1.3.**1**" and "1.3.**2**". The algorithm for finding the moved ancestors is depicted in Figure 6(a).

After finding the moved ancestors, we are able to find the relative updated leaf elements. The SQL query shown in Figures 3(d) is used to detect the relative updated leaf elements. The SQL query is encapsulated in function `findTheRelative UpdatedLeafNodes` (line 18) in Figure 2(c). The query tries to find the tuples belong to first and second versions of an XML document which have the same local orders (*LocalOrder*), path (*PathID*), but have different value (*Value*) (Figures 3(d), lines 8-10). The prefix of *Dewey* attributes of the tuples corresponding to first and second versions of the XML document are equal to the moved ancestors' *Dewey* in the first and second versions respectively (Figures 3(d), line 11-12). We also have to make sure that the position of the corresponding leaf elements of these tuples are not changed (Figures 3(d), line 13).

After we detect the relative updated leaf elements, we need to delete the corresponding tuples of the updated leaf elements which are in the `T1` and `T2` tables. The deleted rows in the `T1` and `T2` tables in Figure 4(a) are indicated with "$\sharp$" at the right of the rows. The results of this query are stored in the `UR` table as shown in Figure 5(a).

3.5 Absolute Updated Leaf Elements Detection Phase

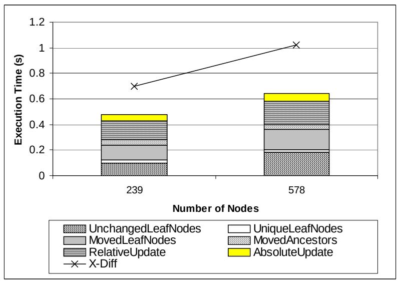
Now, the `T1` and `T2` tables may only consist of the deleted and inserted leaf elements respectively, and the *absolute updated leaf elements* which are detected as a pair of deleted and inserted leaf elements respectively. The objective of this phase is to find the *absolute updated leaf elements*. Intuitively, the *absolute updated leaf elements* are the updated leaf elements which are not the descendants of the moved ancestors. If the prefix of *Dewey* value of a leaf element is equal to the *Dewey* value of a moved ancestor, it must be under that moved ancestor. The *absolute updated leaf elements* are the leaf elements whose corresponding tuples have the same *LocalOrder* values, *PathID* values, *Dewey* values, but have different *Value* values.

Based on the above requirements, we are able to find the *absolute updated leaf elements* by using the SQL query as depicted in Figure 4(b). The SQL query is encapsulated in function `findTheAbsoluteUpdatedLeafNodes` (line 21) in Figure 2(c). After we detect the absolute updated leaf elements, we need to delete the corresponding tuples of these leaf elements in the `T1` and `T2` tables which are already detected as absolute updated leaf elements. The results will be stored in the the `UA` table as depicted in Figure 5(b).

```
Input:
  deweySet: A set of (Dewey1,Dewey2) in table Move
Output:
  movedAncestor: A set of (Dewey1, Dewey2) of the moved ancestors

function findMoveAncestors(deweySet) {
  set movedAncestor to empty;
  while (deweySet is not empty) {
    Data = deweySet.removeFirst();
    DW1 = splitDewey(Data.Dewey1);  // Split "1.2.3.4"
    DW2 = splitDewey(Data.Dewey2);  // to <1,2,3,4>
    // Start checking from the local order of the parent node
    for (i=DW1.length - 2; i>=0; i++) {
      if (DW1[i] != DW2[i]) {
        ancestorDW1 = getAncestorDewey(DW1, i);
        ancestorDW2 = getAncestorDewey(DW2, i);
        if ( (ancestorDW1,ancestorDW2) not in movedAncestor) {
          movedAncestor.addLast((ancestorDW1,ancestorDw2));
          // Add the moved ancestors' Dewey values
          deweySet.addLast((ancestorDW1,ancestorDW2));
          break;
        }
      }
    }
  }
  return movedAncestor;
}
```

(a) Algorithm *findMovedAncestor*



(c) Performance: X-Diff and SUCXENT

Code	Number of Nodes	Size (Kb)
SIGLN-01	239	12
SIGLN-02	578	22
SIGLN-03	1,673	64
SIGLN-04	3,259	124
SIGLN-05	6,358	240

Code	Number of Nodes	Size (Kb)
SIGLN-06	13,569	513
SIGLN-07	27,524	1,040
SIGLN-08	69,098	2,610
SIGLN-09	166,064	6,272

(b) Datasets

Fig. 6. Algorithm, Datasets, and Experimental Results.

3.6 Inserted and Deleted Leaf Elements Detection Phase

After detecting the absolute updated leaf elements, the `T1` and `T2` tables will only consist of the corresponding tuples of deleted and inserted leaf elements respectively. Hence, we do not need an SQL query to detect the inserted and deleted leaf elements.

4 Experimental Results

We now present the results of our experiments to study the performance of our database approaches for detecting changes on ordered XML documents.

4.1 Experimental Setup and Data Set

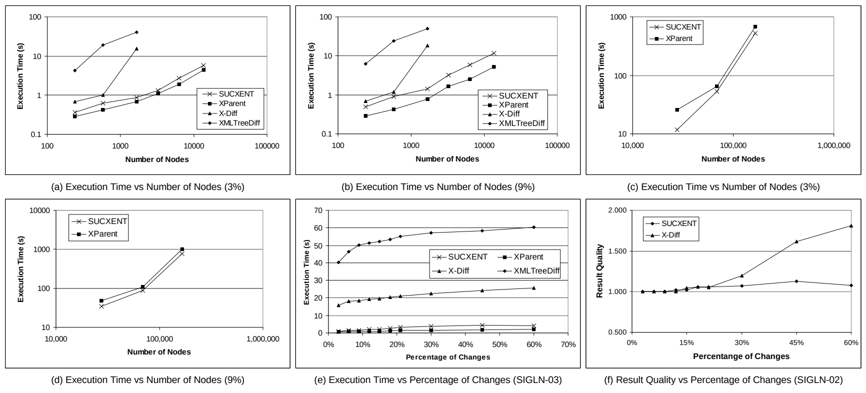
We have implemented our approach entirely in Java. The Java implementation and the database engine were run on a Microsoft Windows 2000 Professional machine with a Pentium 4 1.7 GHz with 512 MB of memory. The database system we used was IBM DB2 UDB 8.1. We also created appropriate indexes on the relations to expedite query processing. We used a set of synthetic SIGMOD Record XML documents. Figure 6(b) shows the characteristics of our data sets. Note that we focus on numbers of nodes in our data sets. The higher number of nodes in a tree will increase the query cost since the database engine will join more tuples. We generated the second version of each XML document by using our own change generator.

We also studied the performance of the state-of-the-art approaches. We could not find the Java version of XyDiff [2][1] which was proposed to solve the problem of detecting changes on ordered XML documents. Hence, we compared our approach to X-Diff [1][2] and XMLTreeDiff [3][3] which are implemented in Java. Since X-Diff was proposed for detecting changes in unordered XML documents, our change generator did not permute the order of any nodes.

[1] Unfortunately, despite our best efforts (including contacting authors), we could not get the Java version.

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

[3] Available at http://www.alphaworks.ibm.com



(a) Execution Time vs Number of Nodes (3%) (b) Execution Time vs Number of Nodes (9%) (c) Execution Time vs Number of Nodes (3%)

(d) Execution Time vs Number of Nodes (9%) (e) Execution Time vs Percentage of Changes (SIGLN-03) (f) Result Quality vs Percentage of Changes (SIGLN-02)

Fig. 7. Datasets and Experimental Results.

We also implemented our approach using XParent schema [5] by doing some modifications. XParent outperforms some model-mapping approaches in some cases as far as path expression queries are concerned. We add one more relation for storing the information on the documents. The `Document` table consists of two tuples: *DocID* and *DocuName* which records the unique id and filename of XML documents respectively. Hence, XParent is able to store multiple versions of XML documents. We notice that the `Data` table in XParent stores information on leaf elements, while the `Element` table stores information on all nodes (including leaf elements). We modify such that the `Element` table only stores information on internal elements in order to avoid data redundancy. We use pre-order traversal rather than level-order traversal for node id in XParent because we use SAX Parser that works in depth-first search fashion. We also added attribute *Dewey*, that has the same function as that in modified SUCXENT schema, in the `Element` and `Data` tables. Due to space constraints, we do not show the SQL queries for detecting changes using XParent.

4.2 Execution Time vs Number of Nodes

In this experiment, we analyze the execution time of our approach implemented in SUCXENT and XParent on different numbers of nodes.

Figures 7(a) and (b) depict the execution times of SUCXENT, XParent, X-Diff, and XMLTreeDiff on small datasets (the number of nodes is less than 20,000 nodes) when the percentages of changes are set to 3% and 9% respectively. We notice that XParent has better performance compared to other approaches. XParent is about 2 to 25 times faster than X-Diff, and is about 15 to 70 times faster than XMLTreeDiff. Compared to SUCXENT, XParent performs better in terms of execution time. XParent is about 1.2 to 2 times faster than SUCXENT for small datasets. Figures 7(c) and (d) depict the execution times of SUCXENT, and XParent on large datasets (the number of nodes is more than 20,000 nodes) when the percentages of changes are set to 3% and 9% respectively. We notice that SUCXENT is about faster 1.3 to 2 times faster than XParent for large datasets. We also notice that SUCXENT and XParent have better scalability and performance compared to X-Diff and XMLTreeDiff. X-Diff and XMLTreeDiff fail to detect large XML documents due to lack of main-memory. In this experiment, X-Diff and XMLTreeDiff are only able to detect the changes on XML documents which have around 2000 nodes.

The difference of execution time between SUCXENT and X-Diff increases as number of nodes increased. Recall the complexity of X-Diff as discussed in [1]. The performance of X-Diff is significantly influenced by the number of nodes and the out-degrees of nodes. X-Diff performs more numbers of node comparisons when the number of nodes and out-degree of nodes are increased. That is, the increment of the numbers of nodes from 400 to 2000 significantly influences the performance of X-Diff. In the first two phases of our approach, we query from the `LeafValue` table. The query plans of these queries show that RDBMS uses the indexed access method. In other phases, we only query from the `T1`, `T2`, `UNCHANGED`, `MOVE`, `UR`, and `UA` tables that are relatively much smaller than the `LeafValue` table. We also create appropriate indexes on these tables to expedite query processing. Figure 6(c) depicts the comparison between X-Diff and each phase in our approach using the first two datasets. We observe that most of the execution time of our approach is taken by finding the unchanged leaf nodes, detecting moved leaf nodes, and detecting the relative updated leaf nodes. Even then, it is faster than X-Diff.

The execution times of first and second steps (*unchanged leaf elements detection* phase and *potential deleted and inserted leaf elements detection* phase respectively) of SUCXENT and XParent are significantly different. The execution times of other steps are same because we use the same tables and the same SQL queries. XParent is faster than SUCXENT for small XML documents because XParent does not perform θ-joins while SUCXENT does. When the number of nodes is increased, the performance of XParent becomes slower than SUCXENT because the size of the `Ancestor` table in XParent becomes very large. Consequently, the cost of θ-joins performed by SUCXENT is cheaper that one of joining the `Ancestor` table in XParent.

4.3 Execution Time vs Percentage of Changes

In this experiment, we analyze the execution time of our approach implemented in SUCXENT and XParent on different percentages of changes.

Figure 7(e) depicts the execution times of SUCXENT, XParent, X-Diff, and XMLTreeDiff on different percentages of changes. We use dataset "SIGLN-03" in this set of experiments. We observed that the execution times of SUCXENT, XParent, and X-Diff are affected insignificantly to the changes of percentages of changes. When we vary the percentages of changes from "3%" to "60%", the execution times of SUCXENT and XParent are less than 5 seconds. If the percentages of changes are increased from "3%" to "60%", the execution times of X-Diff are between 15 and 25 seconds. We also notice that the execution times of XMLTreeDiff are affected to the changes of percentages of changes. The execution times of XMLTreeDiff are between 40 and 60 seconds if the percentages of changes are increased from "3%" to "60%".

4.4 Result Quality vs Percentage of Changes

In this set of experiments, we study the result quality [1] of our approach compared to the X-Diff. The results of SUCXENT and XParent are same because we use the same queries against the same tables. In X-Diff, we may have insertions and deletions of subtrees. In this case, we only count numbers of leaf elements which are in the inserted and deleted subtrees. We use dataset “SIGLN-02” in this set of experiments.

Figure 7(f) depicts the result quality of our approach compared to X-Diff of different percentages of changes. We notice that if the percentage of changes is less than 10%, both approaches results the optimal deltas. When the percentage of changes becomes larger, our approach has better quality results compared to the X-Diff. This happens because when the XML documents are changed significantly, some subtrees in the XML trees are also changed significantly. Hence, X-Diff may detect these subtrees as inserted and deleted subtrees. Note that X-Diff does not focus only on the leaf elements changes.

5 Conclusions

In this paper, we present an approach to detect the content changes on ordered XML documents stored in relational databases. This approach focuses on the changes to the leaf elements. This approach is motivated by the scalability problem on the native approaches and the necessity of detecting the changes on the leaf elements for several applications. In our approach, first, it tries to find the *unchanged*, *potential* inserted, and *potential* deleted leaf elements from the XML documents. These leaf elements are kept in tables. From these tables, the SQL queries are issued in order to find the *moved*, the *relative updated*, and *absolute updated* leaf elements. The experimental results indicate that our approach has better performance and scalability compared to the native approaches. We also show that our approach has better result quality for the XML documents which are changed significantly. We also notice that the performance of our approach is schema dependent, while the result quality of our approach is schema independent.

References

1. Y. Wang, D. J. DeWitt, J. Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. *Proceedings of 19th ICDE, India*, 2003.
2. G. Cobena, S. Abiteboul, A. Marian. Detecting Changes in XML Documents. *Proceedings of 18th ICDE, San Jose, California, USA*, 2002.
3. Curbera, D. A. Epstein. Fast Difference and Update of XML Documents. *XTech'99, San Jose*, 1999.
4. D. Florescu, D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin, Volume 22, Number 3*, 1999.
5. H. Jiang, H. Lu, W. Wang, J. Xu Yu. XParent: An Efficient RDBMS-Based XML Database System. *Proceedings of the 18th ICDE 2002 (Poster Paper), San Jose, California, USA*, 2002.
6. Online Computer Library Center. Introduction to the Dewey Decimal Classification. *http://www.oclc.org/oclc/fp/about/about_the_ddc.htm.*
7. S. Prakash, S. S. Bhowmick, S. Mardia. SUCXENT: An Efficient Path-based Approach to Store and Query XML Documents. *Proceedings of the 15th DEXA, Zaragoza, Spain*, 2004.