Chapter 8
Detecting the Changes to Hybrid XML

In the prior chapters, we have elaborated on our approaches for detecting the changes to ordered and unordered XML documents. Recall that we have three different types of XML documents, that is, ordered XML, unordered XML, and hybrid XML. In this chapter, we shall discuss our works related to hybrid XML. We shall divide our discussion into two parts. In the first part, we shall elaborate on XANDY-H that is a variant of XANDY for detecting the changes to hybrid XML documents. The second part is about detecting the changes to DTDs. In many applications a schema (i.e., Document Type Definition (DTD) or XML schema (XSD) [Sch]) is associated with a set of XML documents to define their legal structures. Often DTDs contain sequence (denoted by “;”) and choice (denoted by “|”) groups. The order of elements in a sequence group is important, while the order of elements in a choice group is not significant. In this second part, we present DTD-DIFF, an algorithm for detecting the changes to DTDs. Note that the preliminary works on XANDY-H and DTD-DIFF can be found in [LBB05] and [LHBM06], respectively.

This chapter is organized as follows. Section 8.1 discusses XANDY-H, the relational-based approach for detecting the changes to hybrid XML documents. In Section 8.2, we elaborate on DTD-DIFF, an approach for detecting the changes to DTDs. Finally, we summarize this chapter in Section 8.3.

8.1 Detecting the Changes to Hybrid XML

In this section, we shall present a relational-based approach for detecting the changes to hybrid XML documents called XANDY-H. First, we shall present modification of relational
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

Input:
- \( n \): number consecutive hybrid XML documents in RDBMS
- \( \text{threshold} \): order threshold

Output:
- a `nodeType` table

1. `findOrderedXXDeltas(n);`
   ```
   /* Rule 1 */
   for all nodes \( m \) that are moved among siblings do
   mark parent(\( m \)) as ordered node;
   end for
   /* Rule 2 */
   /*Find the number of child nodes for each internal nodes. Note that deletions cause the number of child nodes reduced. */
   getNumberOfChildNodes();
   /* Analyze the inserted nodes */
   for all inserted nodes \( i \) do
   if (\( i \) is not inserted as the last child node) then
   add parent(\( i \)).counter;
   add parent(\( i \)).total;
   end if
   end for
   /* The parent nodes are grouped by their paths */
   groupParentNodesByPath();
   /* Calculate the proportions */
   for each group \( g \) that has path \( p \) do
   \( g.\text{counter} = \text{sum}(\text{parent}(i).\text{counter}); \)
   \( g.\text{total} = \text{sum}(\text{parent}(i).\text{total}); \)
   \( g.\text{score} = \frac{g.\text{counter}}{g.\text{total}}; \)
   if (\( g.\text{score} > \text{threshold} \)) then
   mark path \( p \) as path of ordered nodes
   end if
   end for
   /* Store the result */
   Store in `nodeType` table
   ```

21. Store in `nodeType` table

Figure 8.1: XANDY–H: The `findNodeType` Algorithm.

---

<table>
<thead>
<tr>
<th>Path</th>
<th>Name</th>
<th>Level</th>
<th>Score</th>
<th>Status</th>
<th>Path_ID</th>
<th>Level</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>books</code></td>
<td>books</td>
<td>1</td>
<td>0.7520</td>
<td>unordered</td>
<td>1</td>
<td>3</td>
<td>unordered</td>
</tr>
<tr>
<td><code>books/book</code></td>
<td>book</td>
<td>2</td>
<td>0.5780</td>
<td>unordered</td>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td><code>books/book/authors</code></td>
<td>authors</td>
<td>3</td>
<td>0.8170</td>
<td>ordered</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td><code>books/book/chapters</code></td>
<td>chapters</td>
<td>3</td>
<td>0.8920</td>
<td>ordered</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) NodeType Table  (b) CNodeType Table

Figure 8.2: XANDY–H: The `nodeType` Table.

schema used for storing hybrid XML documents before the changes are detected. Note that XANDY–H uses SUCXENT for storing hybrid XML documents. Next, we shall discuss our order learning module that is used to determine automatically the ordered and unordered nodes by analyzing the changes to a set of consecutive versions of a hybrid XML document. As XANDY–H is a variant of XANDY, it also consists of two phases: the finding best matching subtrees phase and the detecting the changes phase. We shall elaborate on each phase in turn. To the best of our knowledge, XANDY–H is the first published approach that addresses the problem of detecting the changes to hybrid XML documents.

### 8.1.1 Background

The modifications of SUCXENT schema in XANDY–H are similar to the ones in XANDY–O. The modified SUCXENT schema is depicted in Figure 4.1(b). The difference is only on the way we store child nodes of the unordered nodes. For the child nodes of the unordered nodes, we do not store their left-to-right position among their siblings. Consequently, the value of the `LocalOrder` attribute in the `LeafValue` and `AncestorInfo` tables will be equal to "null". On the other hand, we need to keep local orders of child nodes of the ordered nodes. They are stored in the `LocalOrder` attribute.
8.1.2 Ordered Learning Module

In order to detect the changes to hybrid XML, we first must determine automatically the ordered and unordered nodes by analyzing the changes to a set of consecutive versions of a hybrid XML document. We use the following two heuristics to facilitate identification of ordered nodes.

(i) Let node $x$ be the $i$th child of node $p$ in version $v_1$. Then, if $x$ is moved from the $i$th to the $j$th position in version $v_2$ where $j \neq i$ then $p$ is an ordered node.

(ii) Let $x_1, x_2, ..., x_k$ be a set of nodes inserted as children of node $p$ in versions $v_1, v_2, ..., v_m$. Let $n$ be the number of nodes that are not inserted as right-most or left-most child of $p$ where $0 < n \leq k$. Then $p$ is an ordered node if $n/k \geq r$ where $r$ is called the order threshold.

Figure 8.1 depicts the algorithm used to find the type of the nodes in the hybrid XML documents. Given a sequence of consecutive versions of a hybrid XML document stored in RDBMS and the order threshold, the findNodeType algorithm first detects the changes to these XML documents (line 1). As we need to know whether there are moved nodes, the algorithm uses an ordered change detection algorithm [LB] to detect the changes. Next, the algorithm analyzes the moved nodes that are moved among their siblings from the changes (Rule 1, lines 2-4). If node $m$ is moved among its siblings, then $parent(m)$ is marked as ordered nodes. Next, the algorithm starts analyzing the inserted nodes (Rule 2, lines 5-20). The algorithm calculates the number of child nodes of the internal nodes (line 5). Note that it also considers the deletions. Hence, a deletion of a child node shall reduce the number of child nodes by one. The next step is to calculate the number of inserted nodes that are not inserted as last child nodes (lines 6-20). The parent nodes of the inserted nodes are grouped by their node names (line 12). Then, the algorithm calculates the proportion of number of insertions occurred at the positions other than the last position and total number of insertions to a particular internal node (lines 13-20). If this proportion is greater than the order threshold, then all internal nodes that have the same path as the path of this particular node are considered as ordered nodes. Finally, the result will be stored in the nodeType table (Figure 8.2(a)).
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

8.1.3 Phase 1: Finding Best Matching Subtrees

In this section, we shall discuss Phase 1 in XANDY–H that is used to determine the best matching subtrees in $T_1$ and $T_2$. Note that we are able to use similar terminology as defined in Chapter 3. The difference is in calculating the similarity score. The similarity score $\mathcal{R}$ of two subtrees $t_1$ and $t_2$ as follows.

(i) If $t_1$ and $t_2$ are the unordered nodes, then $\mathcal{R}(t_1, t_2) = \frac{2|A|+2|B|}{|t_1 \cup t_2|} = \frac{2|t_1 \cap t_2|}{|t_1 \cup t_2|}$.

(ii) If $t_1$ and $t_2$ are the ordered nodes, then $\mathcal{R}(t_1, t_2) = \frac{2|A|+|B|}{|t_1 \cup t_2|}$,

where $|t_1 \cup t_2|$ is the total number of leaf nodes of subtrees $t_1$ and $t_2$, $|t_1 \cap t_2|$ is number of matching leaf nodes, and $|A|$ and $|B|$ are numbers of nodes of fixed and shifted matching leaf nodes in $t_1$ and $t_2$ respectively ($A \cap B = \emptyset$).

The findBestMatchingSubtrees algorithm in XANDY–H is similar to the one in XANDY–O as discussed in Chapter 4 (without line 9). Note that we do not need to populate the best matching subtrees because of the same reasons presented in Chapter 6. The findBestMatchingSubtrees algorithm consists two sub phases: the finding matching sibling orders phase and the bottom-up matching phase.
Finding Matching Sibling Orders. Before we start finding the matching sibling orders, we have a pre-processing step. This is used to generate the $C_{Node\text{Type}}$ table (Figure 8.2(b)). Basically, the $C_{Node\text{Type}}$ table contains the distinct path of leaf nodes. The $Status$ attribute is used to define whether a leaf node is the child node of ordered or unordered nodes. For example, the first tuple in the $C_{Node\text{Type}}$ table (Figure 8.2(b)) explains that all leaf nodes with $pathId=1$ are the child nodes of an unordered node. Similarly, the second tuple defines that all leaf nodes with $pathId=2$ are the child nodes of an ordered node.

The $findMatchingSiblingOrder$ algorithm that is used for finding matching sibling orders is depicted in Figure 8.3(a). The algorithm works as follows. First, it determines the matching leaf nodes that are the child nodes of unordered nodes (line 1) by using SQL query depicted in Figure 8.3(b). Observe that this SQL query is similar to the one used in XANDY-U. The difference is that we include the $C_{Node\text{Type}}$ table in joins. This is important as in this step we only need to find the matching leaf nodes that are the child nodes of unordered nodes. We store the result of the SQL query depicted in Figure 8.3(b) in the TempLV table. The structure and semantics of the TempLV table are similar to the ones presented in Chapter 6. Similar to XANDY-U, we also need to refine the detected matching leaf nodes that are the child nodes of unordered nodes (line 2, Figure 8.3(a)). This is because of the same reasons as discussed in Chapter 6.

The next step is to find the matching leaf nodes that are the child nodes of ordered nodes (lines 3-4, Figure 8.3(a)). We determine the fixed and shifted matching leaf nodes by using SQL queries depicted in Figures 8.3(c) and 8.3(d). These SQL queries are extended from the SQL queries depicted in Figures 4.13(a) and 4.13(b). Lines 12–13 (Figure 8.3(c)) and line 10 (Figure 8.3(d)) are used to guarantee that matching leaf nodes are the child nodes of ordered nodes. We store the detected fixed and shifted matching leaf nodes in the FixedLV and ShiftedLV tables, respectively. The structure and semantics of the FixedLV and ShiftedLV tables are similar to the ones presented in Chapter 4.

Having determined the matching leaf nodes, we are ready to find the matching sibling orders. First, we find the matching sibling orders from matching leaf nodes that are the child nodes of unordered nodes (line 5, Figure 8.3(a)). We use the same SQL query as depicted in Figure 6.5(d). Next, we determine the matching sibling orders from the fixed and shifted matching leaf nodes (lines 6-9, Figure 8.3(a)). We use the similar SQL queries that are used by XANDY-O (Figures 4.13(c)–4.13(f)). Finally, we need to update the value
of the Total attribute as they are still equal to ‘0’. Figure 4.13(g) is used to update the value of the Total attribute.

**Bottom-up Matching.** The bottom-up matching phase in XANDY–H is similar to the one in XANDY–O. Instead of repeating the discussion, we shall present the difference between the bottom-up matching phase in XANDY–H and the one in XANDY–O. The bottom-up matching phase in XANDY–O can be found in Chapter 4.

The difference is in the maximizeSimilarityScore function. In XANDY–O, if $i_x \neq i_y$ but $parent(i_x) \neq parent(i_y)$, then we annotate the corresponding tuple of $i_x \neq i_y$ in the Matching table. In XANDY–H, we use the maximizeSimilarityScore function as in XANDY–U. That is, if $i_x \neq i_y$ but $parent(i_x) \neq parent(i_y)$, then we delete the corresponding tuple of $i_x \neq i_y$ in the Matching table. This is because we do not detect the moves to other parent nodes. The structure and semantics of the Matching table are similar to the ones presented in Chapter 4. We also need to add the Name attribute in which node names of matching internal nodes are stored.

### 8.1.4 Phase 2: Detecting the Changes

After the best matching subtrees are determined, we are ready to detect the changes to hybrid XML documents. Figure 8.4 depicts the algorithm used for detecting the changes. In this section, we shall elaborate briefly on these sub phases used to detect different types of changes.

<table>
<thead>
<tr>
<th>Input: did1 and did2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: XDelta tables</td>
</tr>
<tr>
<td>1 detectInsertedInternalNodes (did1, did2);</td>
</tr>
<tr>
<td>2 detectDeletedInternalNodes (did1, did2);</td>
</tr>
<tr>
<td>3 detectInsertedLeafNodes (did1, did2);</td>
</tr>
<tr>
<td>/* Detecting absolute updated leaf nodes that are child nodes of ordered nodes */</td>
</tr>
<tr>
<td>4 detectAbsoluteUpdate (did1, did2);</td>
</tr>
<tr>
<td>/* Detecting relative updated leaf nodes that are child nodes of ordered nodes and updated leaf node that are child nodes of unordered nodes */</td>
</tr>
<tr>
<td>5 detectRelativeUpdate (did1, did2);</td>
</tr>
<tr>
<td>6 updateCorrector (did1, did2);</td>
</tr>
<tr>
<td>7 detectMoveAmongSiblings (did1, did2);</td>
</tr>
</tbody>
</table>

Figure 8.4: XANDY–H: The detectTheChanges Algorithm.
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

Figure 8.5: XANDY–H: The Delta Tables.

Insertion and Deletion of Internal Nodes

The SQL query in XANDY–H for detecting the inserted/deleted internal nodes is similar to the ones in XANDY–O. The SQL query for detecting the insertion of internal nodes is depicted in Figure 4.34(a). The one for detecting the deleted internal nodes are similar to the one depicted in Figure 4.34(a) after slight modifications. We change did2 in line 6 to did1. The MinSO2 and MaxSO2 are replaced by MinSO1 and MaxSO1, respectively. The inserted/deleted internal nodes that are the child nodes of unordered nodes can also be detected by using this SQL query. However, the value of the LocalOrder attribute will be null as we do not store the left-to-right positions of nodes that are the child nodes of unordered nodes.

Consider two tree representations of hybrid XML documents shown in Figure 1.10. We notice that the subtree rooted at node 102 in T2 is inserted. Subtree t11 that is rooted at node 11 is a deleted subtree. The inserted internal nodes are retrieved by the SQL query depicted in Figure 4.34(a) and stored in the INS_INT table as shown in Figure 8.5(a). The DEL_INT table stores the deleted internal nodes. It is depicted in Figure 8.5(b). The schema and semantics of attributes of the INS_INT and DEL_INT tables in XANDY–H are similar to...
the ones in XANDY-O. They are depicted in Figure 4.35.

**Insertion and Deletion of Leaf Nodes**

Similarly, the SQL query in XANDY-H for detecting the inserted/deleted leaf nodes in are similar to the ones in XANDY-O. We use two SQL queries as depicted in Figures 4.34(b) and 4.34(c). For detecting the deleted leaf nodes, we also use two SQL queries as depicted in Figures 4.34(b) and 4.34(c) after slight modifications. We change did2 in line 6 (Figure 4.34(b)) and in lines 9 and 21 (Figure 4.34(c)) to did1. We replace INS_INT in line 4 (Figure 4.34(b)) with DEL_INT. The did1 in line 17 (Figure 4.34(c)) is replaced by did2. The MinSO2 and MaxSO2 in lines 11 and 22 (Figure 4.34(c)) are replaced by MinSO1 and MaxSO1, respectively. We replace MinSO1 and MaxSO1 in line 19 (Figure 4.34(c)) with MinSO2 and MaxSO2, respectively. Note that the updated leaf nodes are also detected in this phase as they can be decomposed as pairs of deleted and inserted leaf nodes.

Consider two tree representations of hybrid XML documents shown in Figure 1.10. We notice that node 116 is an inserted leaf node that is in best matching subtrees. Node 103 is one of inserted leaf nodes that are in a newly inserted subtree. Node 12 is a deleted leaf node that is in a deleted subtree. The INS_LEAF (Figure 8.5(c)) and DEL_LEAF (Figure 8.5(d)) tables are used to store the inserted and deleted leaf nodes, respectively. The highlighted tuples in Figures 8.5(c) and 8.5(d) are the corresponding tuples of updated leaf nodes. The schema and semantics of attributes of the INS_LEAF and DEL_LEAF tables in XANDY-H are similar to the ones in XANDY-O. They are depicted in Figure 4.35.

**Content Updates of Leaf Nodes**

Intuitively, an updated node is available in the first and second versions, but its value is different. Update operations on the leaf nodes depend on the node types of their parents. If
the parent nodes are ordered nodes, then update operations on the leaf nodes can be classified into absolute updates and relative updates. In the absolute update, the node’s position in the DOM tree is not changed, but the value has changed. In the relative update operation, the absolute position as well as the value of the node has changed due to insert/delete/move operations on other nodes. Otherwise, we only have one kind of update operation that is similar to relative update operation. We detect the updated leaf nodes by using the \texttt{INS\_LEAF} and \texttt{DEL\_LEAF} tables in which the inserted and deleted leaf nodes are stored respectively. In addition, we also need to use the \texttt{Matching} table in order to guarantee that the updated leaf nodes are in the matching subtrees. Note that we only consider the update of the content of the leaf nodes. Similar to [WDyC03], the modification of the name of an internal node is detected as a pair of deletion and insertion.

**Absolute Updated Leaf Nodes.** Figure 8.6(a) is used to detect the absolute updated leaf nodes. Note that the intuition of this SQL query is similar to the one in XANDY-O. The difference is that in XANDY-H we need to specify a condition that indicates that the parent nodes must be ordered nodes. Line 7 is used to specify that the parent nodes of updated leaf nodes are ordered nodes. Lines 10 and 11 are according to Definition 3.11 (for absolute updates). In line 4, we set a value for the \texttt{UPD\_TYPE} attribute. This attribute is to indicate whether a tuple is a corresponding tuple of an absolute or relative updated leaf node. The absolute updated leaf nodes are stored in the \texttt{UPD\_LEAF} table (Figure 8.5(e)). The schema and semantics of attributes of the \texttt{INS\_LEAF} and \texttt{DEL\_LEAF} tables in XANDY-H are similar to the ones in XANDY-O. They are depicted in Figure 4.35. The difference is that in XANDY-H we have \texttt{UPD\_TYPE} attribute. After we detect an absolute updated leaf node $u_a$, we need to delete the corresponding tuple of a deleted leaf node $e_a$ in the \texttt{DEL\_LEAF} table and one of an inserted leaf nodes $y_a$ in the \texttt{INS\_LEAF} table.

**Relative Updated Leaf Nodes.** The relative updated leaf nodes (if the parent nodes are ordered nodes) and the updated leaf nodes (if the parent nodes are unordered nodes) can be detected by using SQL query depicted in Figure 8.6(b). The intuition behind this SQL query is to find the leaf nodes whose parent nodes are best matching internal nodes (lines 12–14) and that have the same path (line 10) but different node values (line 11). We do not need to specify the node type of parent nodes as we have deleted the detected absolute updated leaf nodes. That is, what left are the relative updated leaf nodes (if the parent nodes are ordered nodes) and the updated leaf nodes (if the parent nodes are unordered nodes). Recall
that we may have incorrect results when we detect the updated leaf nodes in \textsc{Xandy}–\textsc{U} and the relative updated leaf nodes in \textsc{Xandy}–\textsc{O}. We also face the same problem in \textsc{Xandy}–\textsc{H}. Therefore, we need to correct the result of SQL query depicted in Figure 8.6(b) by using the \textit{updateCorrector} algorithm. The \textit{updateCorrector} algorithm in \textsc{Xandy}–\textsc{H} is similar to the one in \textsc{Xandy}–\textsc{O}. The updated leaf nodes are stored in the UPDXEAF table. After we detect a relative updated leaf node \(u_r\), we also need to delete the corresponding tuple of a deleted leaf nodes \(e_r\) in the DEL\_LEAF table and one of an inserted leaf nodes \(y_r\) in the INS\_LEAF table.

\textbf{Move Among Siblings}

Detecting the nodes that are moved among their sibling nodes in \textsc{Xandy}–\textsc{H} is similar to the one in \textsc{Xandy}–\textsc{O} as discussed in Chapter 4. The difference is that \textsc{Xandy}–\textsc{H} only detects the moved nodes that are the child nodes of ordered nodes. Consequently, we need to modify the SQL queries for generating the moveList table. Note that they are depicted in Figures 4.43(c) and 4.43(d). The modifications are as follows. We add “\textit{NODETYPE AS NT}” in the FROM-clause of SQL queries depicted in Figures 4.43(c) and 4.43(d). We also insert “\textit{NT.Status=‘ordered’ AND C.Level = NT.LEVEL AND C.Name = NT.Name}” in the WHERE-clause.

The nodes that are moved among their siblings are stored in the moveList table. In our example, the moveList table is shown in Figure 8.5(f). The tuples keep the move information of nodes 5 and 6. Observe that nodes 5 and 6 are moved among their siblings.

\textbf{8.1.5 Experimental Results}

We have implemented \textsc{Xandy}–\textsc{H} entirely in Java. The implementation and the database engine were run on a MS 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 8.7). 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. XANDY has three variants: XANDY-U [LBM05] for unordered XML, XANDY-O [LB] for ordered XML, and XANDY-H [LBB05] for hybrid XML. We compared the performance of these three variants. We also compared our approach to X-Diff [WDyC03] and X-Diff-O (The option ‘-o’ of X-Diff is activated so it calculates the minimum editing distance in finding the matchings.). Note that despite our best efforts (including contacting the authors), we could not get the Java version of XyDiff [CAM02].

**Result Quality.** In this experiment, we study the effect of the percentage of changes on the result quality by using “Sigmod-03” data set. We set “authors” as the ordered node. A series of new versions are generated by varying the percentage of the changes from “3%” to “30%”. The number of nodes involved in the deltas is counted for each approach. The number of nodes in the optimal XDeltas is compared to the one detected by the different approaches. The ratios are plotted in Figure 8.8(a). We observed that XANDY-H is able to detect the optimal or near optimal deltas. The ratios of XANDY-O and XyDiff are always greater than 1. This is because the movements among siblings of the child nodes of unordered nodes are detected as changes. In XANDY-H, such changes are not detected. The ratios of XANDY-U and X-Diff-O are always less than 1. This is because the unordered approaches do not detect the movements among siblings of the child nodes of ordered nodes. The ratio of X-Diff is less than 1 if the percentage of changes is less than “20%”. Otherwise, it is greater than 1. This is because X-Diff does not calculate the minimum editing distance. Therefore, X-Diff may detect as a deletion of a subtree if it is changed significantly.

**Accuracy of Learning Module.** We generate 30 versions of a hybrid XML document based on SIGMOD DTD. The first version has 645 nodes. We set nodes “articles” and “authors” as ordered nodes, and \( \tau \) is set to “0.75”. Figure 8.8(b) depicts the accuracy of the learning module. The ordered proportion of “articles” is equal to 1 and detected as ordered nodes for the first 6 versions. The insertion of node “article” only occurs once. This insertion does not happen at the last position. After analyzing more than 18 versions, the learning module determines nodes “articles” and “authors” as ordered nodes as the ratios are greater than \( \tau \). The accuracy of the learning module depends on the changes.
occur in the XML documents. The learning module may fail to detect the ordered nodes in certain cases. For example, node "authors" is not detected as an ordered node if there is no occurrence of the move operation and the insertions occurred at the positions other than the last position rarely happen.

**Execution Time vs. Number of Nodes.** The percentages of changes are set to "3%" and "9%" and the threshold \( \theta \) is set to "0.0" which shall give us the upper bound of the execution time. We set "authors" as the ordered node.

Figures 8.9(a) and 8.9(d) depict the performance of Phase 1 in \textsc{Xandy-U}, \textsc{Xandy-H} and \textsc{Xandy-O}. \textsc{Xandy-U} is faster than \textsc{Xandy-H} and \textsc{Xandy-O}. This is because \textsc{Xandy-U} does not distinguish between fixed and shifted matching leaf nodes. We observe that \textsc{Xandy-H} is slightly faster than \textsc{Xandy-O}. This is because of the following reason. Recall that \textsc{Xandy-H} determines three different matching leaf nodes: the ones that are the child nodes of unordered nodes, the fixed and shifted matching leaf nodes. In \textsc{Xandy-O}, we only find the fixed and shifted matching leaf nodes. However, in this set of experiments, we only have an ordered node. Consequently, the execution time for finding the fixed and shifted matching leaf nodes in \textsc{Xandy-H} is less dominant than the one for detecting finding the matching leaf nodes that are the child nodes of unordered nodes in \textsc{Xandy-H}. In addition, \textsc{Xandy-H} does not need to populate the best matching subtrees from the TempMatching table. We shall see later the effects of numbers of ordered nodes to the performance of \textsc{Xandy-H}.

Figures 8.9(b) and 8.9(e) depict the performance of Phase 2 in \textsc{Xandy-U}, \textsc{Xandy-H} and \textsc{Xandy-O}. Similarly, we analyze that \textsc{Xandy-U} is faster than \textsc{Xandy-H} and \textsc{Xandy-O}. This is because \textsc{Xandy-U} does not detect the movement of nodes among theirs siblings.
XANDY-H is faster than XANDY-O. This is because XANDY-O detects the movement of nodes to different parent nodes, while XANDY-H does not do so. The overall performances of our approaches and X-Diff are depicted Figures 8.9(c) and 8.9(f). From this set of experiments, we observe the following observations. Our relational-based approaches are more scalable than X-Diff. Note that X-Diff is unable to detect the changes to the XML documents that have over 5,000 nodes due to lack of main memory. For smaller data sets, X-Diff is faster than our approaches, but after the number of nodes is greater than 2500 nodes, our approaches is faster than X-Diff (up to 40 times). In general, we observe that XANDY-U is faster than XANDY-H and XANDY-O, and XANDY-H is faster than XANDY-O.
Next, we study the effect of number of ordered nodes on the performance of XANDY–H. We use the first four data sets and set the percentage of changes to “3%”. Figure 8.10 depicts the performance of XANDY–H for different numbers of ordered nodes. The performance of XANDY–H is affected by the number of the ordered nodes as the difference of the execution time is up to 5%. In conclusion, the number of ordered nodes influences the performance of XANDY–H. This is because the execution time in Phase 1, in particular the execution time in finding the matching leaf nodes, is increased.

8.2 Detecting the Changes to DTDs

In this section, we shall present a novel algorithm, called DTD-DIFF, for detecting the changes to DTDs.

8.2.1 Background

In many applications, a schema (i.e., Document Type Definition (DTD) or XML schema (XSD) [Sch]) is associated with a set of XML documents to define their legal structures. Schema of such XML documents may also need to be updated for various reasons [GMR05, SKC+01, AJEM03]. Systems must be adapted to the real-world changes. Schemas may be initially defined as drafts and are subsequently refined due to changes to the real-world or due to the need of fixing errors in the previous versions. Commercial alliances change and expand. For example, consider the DTD $D_1$ in Figure 8.11(a) at time $t_1$. It may evolve to $D_2$ (Figure 8.11(b)) at time $t_2$ because the university may wish to restructure the information
due to changes in the university administrators’ requirements. Hence, there is a strong need for a tool to detect changes to DTDs. Such change detection tool can be useful in at least following three ways.

- **Maintenance of XML documents.** A DTD change detection tool can be useful for incremental maintenance or revalidation of a set of XML documents when their DTD evolves. Note that many documents can be associated with the same DTD/schema and the brute-force approach of revalidating a complete document is known to be high [RS04]. For instance, let $X$ be a set of XML documents where each document $x_i \in X$ conforms to DTD $D$. Assume that due to mistakes in the initial design, $D$ is modified to $D'$. Consequently, $x_i \in X$ may not conform to $D'$ anymore. Therefore, it is necessary to detect the differences between $D$ and $D'$ (denoted by $\Delta(D,D')$) automatically so that it can be used to transform $x_i \in X$ to $x'_i$ such that $x'_i$ conforms to $D'$. The basic idea is to keep track of the changes made to the DTD and to identify the portion of the DTD/schema that, because of these changes, requires revalidation. The document portions affected by those changes can then be identified and revalidated, thus avoiding a costly revalidation of the whole document.

- **Incremental maintenance of relational schema.** Recently, there has been a substantial research effort in storing and processing XML data using relational databases...
These approaches can be classified into two major categories. In the schema-conscious approach [STZ+99], a relational schema is created based on the DTD/schema of the XML documents. In the schema-oblivious approach [KKN03], a fixed relational schema is used to store XML documents. The basic idea is to capture the tree structure of an XML document. This approach does not require existence of an XML schema/DTD. A DTD change detection tool can be particularly useful for incremental maintenance of the relational schema generated by a schema-conscious approach. This is because if the DTD is changed, then the respective relational schema may also need to be modified. The differences between the old and new versions of a DTD can help us in maintaining the relational schema incrementally.

- **Maintenance of XML access control policies.** Changes to the DTD or XML schema can impact on the access control policies defined on the XML documents. Hence, it is necessary to incrementally maintain the access policies as the DTD changes.

At first glance, it may seem that the DTD change detection problem can easily be addressed by existing change detection tools for XML documents [LB05b, LBM05, LBD04, CAM02, WDyC03]. Specifically, we can first transform two versions of a DTD to XML...
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

Schemas (XSD), that are in XML format, using tools such as Syntex dtd2xs [Syn] and LuMriX dtd2xs [fE]. For example, Figures 8.12(a) and 8.12(b) are the XSD representations of the DTDs in Figures 8.11(a) and 8.11(b), respectively. Then, the changes to the DTDs can be detected using existing XML change detection tools (such as X-Diff [WDyC03] and XyDiff [CAM02]). Although this approach will clearly detect changes, we argue that it may often fail to detect semantically correct and optimal changes. For example, these algorithms may detect that the name of a school element in $D_1$ in Figure 8.11(a) has been updated to $sinfo$ in $D_2$ in Figure 8.11(b) (Figure 8.13). However, this is semantically incorrect! Furthermore, these algorithms are not efficient as far as DTD change detection is concerned as they do not exploit the structure and semantics of the DTDs to improve response time. We shall elaborate on these issues further in the later section.

In summary, the main contributions of this work are as follows.

(i) We present data model to represent the changes to DTDs. By using this data model we are able to detect the changes to DTDs correctly.

(ii) We propose a novel algorithm called DTD-DIFF for detecting the changes to DTDs. To the best of our knowledge, this is the first approach that addresses the DTD change detection problem. The algorithm takes as input two versions of a DTD that are represented using our DTD data model and detects the changes directly without converting them to XSD format.

(iii) Through an extensive experimental study, we show that our approach is 5-325 times faster than X-Diff, a state-of-the-art XML change detection algorithm. Note that in our study, we convert DTDs to XSD files prior to employing X-Diff to detect the changes. We also show that DTD-DIFF is also able to produce optimal or at least near-optimal deltas.

8.2.2 Related Work

To the best of our knowledge, there aren’t any published works on detecting changes to DTDs. In the preceding section, we mentioned that the two versions of a DTD can be first converted to XML schema using tools such as Syntex dtd2xs and and LuMriX dtd2xs, and then we can detect the changes using any existing XML change detection algorithm. Hence, we first compare our approach with existing XML change detection techniques.
XML Change Detection

Recently, a number of techniques for detecting the changes to XML data have been proposed. XyDiff [CAM02] is a main-memory algorithm for detecting the changes in 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 [WDyC03] 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 memory. Consequently, a number of approaches [LB05b, LBM05, LBDM04] have been proposed to address the scalability problem of XML change detection by using relational databases. In contrast to our approach, the above approaches are not designed to detect changes to DTDs. Consequently, even if we employ these algorithms to XSD representations of DTDs, they suffer from the following limitations.

- **Granularity of types of changes.** The above approaches support the following types of edit operations: insert, delete, update, and move operations of nodes in the tree representations of two versions of an XML document. A node in the XML tree represents an element, text or an attribute node. As the data format of a DTD is different from that of an XML document, the types of changes in old and new versions of a DTD are different from the ones that occur in XML documents. In fact, a DTD has
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

A richer variety of edit operations compared to XML documents. For instance, in an XML document an element does not have any cardinality associated with it. However, an element type in a DTD may have a cardinality which may be updated as the DTD evolves. We shall elaborate on different types of change operations in DTDs in the later section.

- Inability to detect changes to both unordered and ordered nodes. Often DTDs contain sequence (denoted by ":") and choice (denoted by "|") groups. The order of elements in a sequence group is important, while the order of elements in a choice group is not significant. The current approaches for detecting the changes to XML documents focus on either ordered XML [LBDM04, CAM02] or unordered XML [LB05b, LBM05, WDyC03]. Hence, if we use existing techniques then it is indeed possible that certain types of changes are not accurately detected. For example, consider the element type declaration information in $D_1$ and $D_2$ in Figures 8.11(a) and 8.11(b), respectively. We observe that elements telp, fax, and website belongs to the choice group. Furthermore, (telp|fax+1website) and address are in the sequence group. The deltas generated by X-Diff and XyDiff, when the documents in Figures 8.12(a) and 8.12(b) are passed as input, are depicted in Figures 8.13(a) and 8.13(b), respectively. Interestingly, the output of XyDiff specifies that the element fax has undergone a move operation (Lines 9-11 and 17-19 in Figure 8.13(b)). However, this is incorrect as fax belongs to the choice group. On the other hand, X-Diff fails to detect the movement of address element (Figure 8.13(a)).

- Detection of semantically incorrect changes. Consider the element type declaration school in $D_1$ and $D_2$ in Figures 8.11(a) and 8.11(b). X-Diff and XyDiff both detect that element name in school is updated to sinfo (lines 6 and 7 in Figures 8.13(a) and 8.13(b), respectively). However, sinfo consists of name, head, website, telp, and fax. Hence, the change detected by these algorithms is semantically incorrect. The correct edit operations should be deletion of name element and insertion of sinfo element.

- Generation of non-optimal edit scripts. We notice that the changes to the cardinality of an element in the DTD can result in generation of non-optimal edit scripts [WDyC03] by X-Diff or XyDiff. For instance, consider the cardinality of element fax in the
element type declaration information in $D_1$ and $D_2$. The cardinality of element fax is updated from + to ?. X-Diff detects it as a result of two edit operations: a deletion followed by an insertion (line 17, Figure 8.13(a)). Similarly, XyDiff also detects it as two edit operations (lines 4-5, Figure 8.13(b)). However, the correct number of edit operations should be one (update of cardinality).

- **Performance bottleneck.** Lastly, existing XML change detection algorithms such as X-Diff are not efficient when they are used to detect changes to DTDs. This is because they do not exploit the structure and semantics of the DTDs to improve response time. As we shall see in the later section, by exploiting such features of DTDs, our DTD change detection algorithm outperforms X-Diff by 5-325 times!

### DTD and XML Schema Evolution

XEM [SKC+01] is an approach which provides XML-centric data and DTD evolution facilities. The authors proposed a set of evolution operators to achieve this. When DTDs are changed, XEM ensures that the existing XML documents still conform to the new DTD. Similarly, when the XML documents are changed, XEM ensures that the changed XML documents still correspond to the specified DTD. In [AJEM03], the authors proposed an approach to manage DTD evolution. The authors also presented 25 DTD changes and defined their semantics by preconditions and post actions such that the new DTD is valid, existing documents conform to the new DTD, and data is not lost if possible. DTD evolution has also been investigated in [BGMT02] where the focus was on dynamically adapting the schema to the structure of most documents stored in an XML data source. Required modifications are deduced by means of structure mining techniques.

The above approaches are different from DTD-DIFF as DTD-DIFF focuses on detecting the changes to DTDs. XEM and the approach in [AJEM03] are used to manage DTD evolution and to ensure that when DTD is changed, the XML documents that conform to this DTD still conform to the new DTD. That is, XEM and the approach in [AJEM03] do not detect the changes.

Guerrini et al. addressed the problem of XML schema evolution in [GMR05]. They proposed a set of atomic evolution primitives to be applied to the basic components of a schema. The authors showed that all required transformation in an XML schema can be expressed through a sequence of primitives in the set. Furthermore, a set of high level evolution primitives that allows complex changes to be expressed in a compact way is proposed. Finally,
they analyze the impact of such primitives on the validity of XML documents known to be valid for the original schema. Our work differs from the above effort as follows. First, we focus on DTDs instead of XML schemas. Second, we address the problem of detecting the changes automatically whereas Guerrini et al. attempts to identify the parts of documents that need to be revalidated after a certain number of changes have occurred on the schema.

8.2.3 DTD Data Model

The DTD Data Model is a simple, flexible model for representing DTDs. In this section, we begin by briefly describing the DTD Data Model. Next, we present the basic change operations used to modify a DTD.

A DTD consists of entity declaration (<!ENTITY ...>), element type declaration (<!ELEMENT ...>), and attribute declaration (<!ATTLIST ...>) that describe entities, elements, and attributes, respectively. For example, consider the DTD $D_2$ in Figure 8.11(b). Lines 1–3, 4–17, and 18 are examples of entity declarations, element type declarations, and attribute declaration, respectively.

**Element Type Declaration (ETD).** In a DTD, XML elements are declared using element type declaration. Each element type declaration has a name and element content. For example, consider the DTD $D_1$ in Figure 8.11(a). The name and the content of the element type `school` in line 5 are `school` and `(name, dean, department*)`, respectively. Formally, ETD is defined as follows:

**Definition 8.1 [Element Type Declaration (ETD)]** An element type declaration is a 2-tuple $E = (N_E, C_E)$, where $N_E$ is the name of the element type $E$, and $C_E = (\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_n)$
is a sequence called element content of \( E \) where \( \gamma_i \) is a regular expression over the element types.

Observe that an element content can be very complex with multiple levels of nesting. For example, \(<! ELEMENT E1 (E1, (E2|E3), (E4?|E5*| (E6,E7)?)*)> \) where \( \gamma_1 = E1 \), \( \gamma_2 = (E2|E3) \), and \( \gamma_3 = (E4?|E5*| (E6,E7)?)* \). We represent the element content \( C_E \) as a content tree \( T_E \). For example, the content trees of element types school and courses in \( D_1 \) and \( D_2 \) are depicted in Figures 8.14(a) and 8.14(b), respectively. Also, consider the element type declaration \(<! ELEMENT E1 (E1, (E2|E3), (E4?|E5*| (E6,E7)?)*)> \). The content tree \( T_{E_1} \) is depicted in Figure 8.14(c). Formally, we define the content tree as follows:

**Definition 8.2 [Content Tree]** Let \( E = (N_E,C_E) \) be an ETD. Then, the content tree of \( C_E \) is a 3-tuple \( T_E = (\text{root}, K, W) \), where

- \( \text{root} \) is the root node of \( T_E \) and label(\( \text{root} \)) = \( N_E \)
- \( K \) is a set of nodes in \( T_E \). A node \( k \in K \) is a 2-tuple \( k = (V_k, \lambda_k) \), where if \( k \) is a leaf node, then \( V_k \) is the name of element in \( C_E \). Otherwise, \( V_k \in \{"\"", "1"\} \). If \( V_k = "\\"\" \) then child nodes of \( k \) must be ordered. Otherwise, child nodes of \( k \) are unordered.
- \( \lambda_k \) is the cardinality of node \( k \) (optional) where \( \lambda_k \in \{"?", "+", "*"\} \).
- \( W \) is a set of edges in \( T_E \).

**Attribute Declaration (AD).** The attribute declaration in a DTD is used to define the attributes of an element. Each AD has a name of element type to which a set of attributes belongs. Each attribute in the attribute set has a name, type, and an optional default value. For example, reconsider \( D_1 \) in Figure 8.11(a). The attribute declaration of element type course is in line 16. The type of data and default value of the attribute code are CDATA and \#REQUIRED, respectively.

**Definition 8.3 [Attribute Declaration (AD)]** An attribute declaration \( A \) is a 2-tuple \( A = (N_A, S_A) \), where \( N_A \) is the name of element type and \( S_A \) is a set of attributes associated with \( N_A \). An attribute \( a \in S_A \) is a 3-tuple \( a = (N_a, Y_a, D_a) \), where \( N_a \) is the name of the attribute \( a \), \( Y_a \in \{"CDATA", "(en1|en2|...)", "ID", "IDREF", "IDREFS", "NMTOKEN", "NMTOKENS", "ENTITY", "ENTITIES", "NOTATION", "xml:"\} \) is the type of data of the attribute \( a \), \( D_a \in \{"value", "#REQUIRED", "#IMPLIED", "#FIXED value"\} \) is the default value of the attribute \( a \).
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

Entity Declaration (ED). Entities are variables used to define shortcuts to common text. Entity references are references to entities. We have two kinds of entities: general entity and parameter entity. Consider DTD $D_2$ as depicted in Figure 8.11(b). Line 1 is an example of parameter entity. An example of general entity is in line 2. Note that we only consider the general entities. This is because the parameter entities automatically replace the entity references. Entities can be declared as internal or external. An internal ED has a name and a replacement text. On the other hand, an external ED has a name, universal resource indicator (URI), and a content notation. For example, in $D_2$ line 2 is an example of an internal ED. The name and replacement text of this entity are univName and "Open University", respectively. Line 3 (Figure 8.11(b)) is an example of an external ED. The name, URI, and content notation are MyScript, "Script1.pl", and "pl", respectively.

Formally, the ED is defined as follows:

**Definition 8.4 [Entity Declaration (ED)]** An internal entity declaration is a 2-tuple $I = (N_I, R_I)$, where $N_I$ is the name of the entity, and $R_I$ is the replacement text of the entity that replaces the entity reference. An external entity declaration is a 3-tuple $J = (N_J, U_J, P_J)$, where $N_J$ is the name of the entity, $U_J$ is a universal resource indicator (URI) of the entity, and $P_J$ is the content notation.

We are now ready to define how a DTD is represented in our DTD data model. Intuitively, one can think of a DTD as a set of ETDs, a set of ADs, and a set of EDs. We define the DTD formally as follows:

**Definition 8.5 [DTD]** A DTD is a 3-tuple $D = (\mathcal{E}, \mathcal{A}, \mathcal{G})$ where $\mathcal{E}$ is a set of Element Type Declarations (ETD) in D, $\mathcal{A}$ is a set of Attribute Declarations (AD) in D, $\mathcal{G}$ is a set of internal and external Entity Declarations (ED). Also, if the numbers of ETDs, ADs, and EDs in a DTD are $\alpha$, $\beta$, and $\gamma$, respectively, then $|\mathcal{E}| = \alpha$, $|\mathcal{A}| = \beta$, and $|\mathcal{G}| = \gamma$.

Types of Changes

We now describe how a DTD is modified. We discuss the types of changes supported by element type declaration, attribute declaration, and entity declaration. We assume that $D_1 = (\mathcal{E}_1, \mathcal{A}_1, \mathcal{G}_1)$ and $D_2 = (\mathcal{E}_2, \mathcal{A}_2, \mathcal{G}_2)$ are the old and new versions of a DTD, respectively. Also, let $\mathcal{E}_i = \{E_{i1}, E_{i2}, \ldots, E_{in}\}$ be a set of ETDs and $N_i = \{N_{i1}, N_{i2}, \ldots, N_{in}\}$ be a set of element names in the ETDs for $i \in \{1, 2\}$.
Chapter 8. Detecting the Changes to Hybrid XML

Types of Changes to Element Type Declaration (ETD). The types of changes to the ETDs are defined as following.

(i) **Insertion of a new ETD.** $E_x = (N_x, C_x)$ is a new ETD iff $N_x \notin N_1$ but $N_x \in N_2$.
    
    For example, `sinfo` (line 7, Figure 8.11(b)) is inserted into $D_2$.

(ii) **Deletion of an ETD.** $E_x = (N_x, C_x)$ is a deleted ETD iff $N_x \in N_1$ but $N_x \notin N_2$.
    
    For example, `dean` (line 10, Figure 8.11(a)) is deleted from $D_1$.

We now describe the types of changes to the element content. As in our DTD model element contents are represented as content trees, we can define the types of changes to element contents in the context of such content trees. Suppose we have two ETDs $E_1 = (N_{E1}, C_{E1})$ and $E_2 = (N_{E2}, C_{E2})$, where $E_1 \in \mathcal{E}_1$, $E_2 \in \mathcal{E}_2$, and $N_{E1} = N_{E2}$. Let $T_{E1} = (\text{root}_1, K_1, W_1)$ and $T_{E2} = (\text{root}_2, K_2, W_2)$ be two content trees of $C_{E1}$ and $C_{E2}$, respectively. The types of changes in the content trees are as follows.

(i) **Insertion of a leaf node.** Let $\text{parent}(k)$ be the parent node of node $k$. Let $w(p, c)$ be an edge connecting node $p$ to node $c$. Node $k = (V_k, \lambda_k)$ is a new leaf node iff $k \notin K_1$, $k \in K_2$, $w(\text{parent}(k), k) \notin W_1$, and $w(\text{parent}(k), k) \in W_2$. For example, consider $T_{\text{school}}$ as depicted in Figures 8.14(a) and 8.14(b). Node `sinfo` is a new node with $V_k = \{\text{sinfo}\}$ and $\lambda_k$ is empty.

(ii) **Deletion of a leaf node.** Node $k = (V_k, \lambda_k)$ is a deleted leaf node iff $k \in K_1$, $k \notin K_2$, $w(\text{parent}(k), k) \in W_1$, and $w(\text{parent}(k), k) \notin W_2$. Consider $T_{\text{school}}$ as depicted in Figures 8.14(a) and 8.14(b). Nodes “name” and “dean” are deleted leaf nodes.

(iii) **Insertion of a subtree.** Let $S_r$ be a subtree rooted at node $r$. Let $\mathcal{K}$ and $\mathcal{W}$ be two sets of nodes and edges in subtree $S_r$, respectively. Subtree $S_r$ is a new subtree iff $r \notin K_1$, $r \in K_2$, $w(\text{parent}(r), r) \notin W_1$, $w(\text{parent}(r), r) \in W_2$, $\forall_{k \in \mathcal{K}} (k_i \notin K_1)$, $\forall_{w \in \mathcal{W}} (w_j \notin W_1)$, $\forall_{k \in \mathcal{K}} (k_i \in K_2)$, and $\forall_{w \in \mathcal{W}} (w_j \in W_2)$.

(iv) **Deletion of a subtree.** Subtree $S_r$ is a deleted subtree iff $r \in K_1$, $r \notin K_2$, $w(\text{parent}(r), r) \in W_1$, $w(\text{parent}(r), r) \notin W_2$, $\forall_{k \in \mathcal{K}} (k_i \in K_1)$, $\forall_{w \in \mathcal{W}} (w_j \in W_1)$, $\forall_{k \in \mathcal{K}} (k_i \notin K_2)$, and $\forall_{w \in \mathcal{W}} (w_j \notin W_2)$.
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

(v) **Move a leaf node.** Node $k_1 = (V_{k1}, \lambda_{k1})$ is moved to be node $k_2 = (V_{k2}, \lambda_{k2})$ iff $k_1 \in K_1$, $k_2 \in K_2$, $w(\text{parent}(k_1), k_1) \in W_1$, $w(\text{parent}(k_1), k_1) \notin W_2$, $w(\text{parent}(k_2), k_2) \notin W_1$, $w(\text{parent}(k_2), k_2) \in W_2$, $V_{k1} = V_{k2}$, and $\text{parent}(k_1) \neq \text{parent}(k_2)$. Consider $\text{T}_{E1}$ and $\text{T}_{E2}$ as depicted in Figures 8.14(c) and 8.14(d), respectively. The node with $V_k = \"E4\"$ and $\lambda_k = \"?\"$ is a moved leaf node.

(vi) **Move a subtree.** Let $K_x$ and $W_x$ be two sets of nodes and edges in subtree $S_{rx}$, respectively. Subtree $S_{r_1}$ is moved to be subtree $S_{r_2}$ iff $r_1 \in K_1$, $r_2 \in K_2$, $w(\text{parent}(r_1), r_1) \notin W_2$, $w(\text{parent}(r_2), r_2) \notin W_1$, $w(\text{parent}(r_2), r_2) \in W_2$, $\forall_{k_1 \in K_1} (k_1 \in K_1)$, $\forall_{w_1 \in W_1} (w_1 \in W_1)$, $\forall_{k_2 \in K_2} (k_2 \in K_2)$, $\forall_{w_2 \in W_2} (w_2 \in W_2)$, $V_{r_1} = V_{r_2}$, and $\text{parent}(r_1) \neq \text{parent}(r_2)$.

(vii) **Update of order.** Let $k_x$ be a leaf/internal node in $K_x$. Let $order(k)$ be the left-to-right position of node $k$ among its siblings. The order of node $k_1$ is updated to be the order of node $k_2$ iff $order(k_1) \neq order(k_2)$, $\text{parent}(k_1) = \text{parent}(k_2)$, $V_{\text{parent}(k_1)} = \"\", and $V_{\text{parent}(k_2)} = \"\"$. Consider $\text{T}_{E1}$ and $\text{T}_{E2}$ as depicted in Figures 8.14(c) and 8.14(d), respectively. The order of node with $V = \"E7\"$ is updated from “2” to “1”. Similarly, the order of node with $V = \"E6\"$ is updated from “1” to “2”. Note that this type of changes is different from the move a leaf node/subtree. In this type of changes, the parent nodes before and after the changes occur are the same. In the move a leaf node/subtree types of changes, the leaf nodes/subtrees have different parent nodes before and after the changes occur.

(viii) **Insertion of Cardinality.** Let $k_1 = (V_{k1}, \lambda_{k1})$ and $k_2 = (V_{k2}, \lambda_{k2})$ be two nodes where $k_1 \in K_1$, $k_2 \in K_2$, $V_{k1} = V_{k2}$, $\lambda_1 = \varnothing$, and $\lambda_2 \neq \varnothing$. Then $\lambda$ is an inserted cardinality iff $\lambda_2 = \lambda$.

(ix) **Deletion of Cardinality.** Let $k_1 = (V_{k1}, \lambda_{k1})$ and $k_2 = (V_{k2}, \lambda_{k2})$ be two nodes where $k_1 \in K_1$, $k_2 \in K_2$, $V_{k1} = V_{k2}$, and $\lambda_1 \neq \varnothing$. Then, the cardinality of $k_1$ is deleted iff $\lambda_2 = \varnothing$.

(x) **Update of Cardinality.** Let $k_1 = (V_{k1}, \lambda_{k1})$ and $k_2 = (V_{k2}, \lambda_{k2})$ be two nodes where $k_1 \in K_1$, $k_2 \in K_2$, $V_{k1} = V_{k2}$, $\lambda_1 \neq \varnothing$, and $\lambda_2 \neq \varnothing$. Then, the cardinality of $k_1$ is updated in $k_2$ if $\lambda_1 \neq \lambda_2$. Consider two $\text{T}_{\text{courses}}$ as depicted in Figures 8.14(a) and 8.14(b). The cardinality of node course is updated from “*” to “+”.

219
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

Note that we do not consider the update of node name for the following reason. Consider the ETDs school in $D_1$ and $D_2$. We cannot consider that the name of element name is updated to sinfo and element dean is deleted as it will lead us to have a delta that is semantically incorrect. On the other hand, suppose we have a “lastname” element whose name is updated to “surname”. DTD-DIFF detects as a deletion of element “lastname” and an insertion of element “surname” as we do not have information of semantic relationships between “lastname” and “surname”. Note that the delta is still correct even though the result quality is reduced. Therefore, we consider the update of node name as a pair of deletion and insertion of a node.

Types of Changes of Attribute Declaration (AD). Let $N_1$ and $N_2$ be the sets of entity names in $A_1$ and $A_2$, respectively. The types of changes in ADs are defined as follows.

(i) **Insertion of a new AD.** $A = (N_A, S_A)$ is a new AD iff $N_A \notin N_1$ and $N_A \in N_2$.

(ii) **Deletion of an ED.** $A = (N_A, S_A)$ is a deleted AD iff $N_A \in N_2$ and $N_A \notin N_1$.

Suppose we have two ADs $A_1 = (N_{A1}, S_{A1})$ and $A_2 = (N_{A2}, S_{A2})$ where $A_1 \in A_1$, $A_2 \in A_2$, and $N_{A1} = N_{A2}$. Let $N_{s1}$ and $N_{s2}$ be the sets of attribute names in $S_{A1}$ and $S_{A2}$, respectively. The types of changes in ADs are defined as follows.

(i) **Insertion of a new attribute.** Attribute $a = (N_a, Y_a, D_a)$ is a new attribute iff $N_a \notin N_{s1}$ and $N_a \in N_{s2}$.

(ii) **Deletion of an attribute.** Attribute $a = (N_a, Y_a, D_a)$ is a deleted attribute iff $N_a \in N_{s1}$ and $N_a \notin N_{s2}$.

(iii) **Update of attribute type.** The attribute type of attribute $a = (N_a, Y_a, D_a)$ is updated in the new version iff $N_a \in N_{s1}$, $N_a \in N_{s2}$, and $Y_a \neq Y_a'$, where $Y_a'$ is attribute type of $a$ in $S_{A2}$.

(iv) **Update of default value.** The default value of attribute $a = (N_a, Y_a, D_a)$ is updated in the new version iff $N_a \in N_{s1}$, $N_a \in N_{s2}$, and $D_a \neq D_a'$, where $D_a'$ are default values of $a$ in $S_{A2}$.

For example, consider $D_1$ and $D_2$ in Figures 8.11(a) and 8.11(b), respectively. The default value of the attribute year of element course is updated from #IMPLIED to #REQUIRED.
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

Note that we do not consider the update of attribute name for the same reason as in the above discussion.

Types of Changes of Entity Declaration (ED). Let \( N_1 \) and \( N_2 \) be the sets of entity names in \( G_1 \) and \( G_2 \), respectively. The types of changes in EDs are defined as follows.

(i) **Insertion of a new ED.** Entity \( G \) is a new ED iff \( N_g \notin N_1 \) and \( N_g \in N_2 \) where \( N_g \) is the name of the entity in \( G \).

(ii) **Deletion of an ED.** Entity \( G \) is a deleted ED iff \( N_g \notin N_2 \) and \( N_g \in N_1 \).

(iii) **Update of replacement text of internal ED.** The replacement text of internal entity \( I = (N_I, R_I) \) is updated to the replacement text of entity \( I' = (N_{I'}, R_{I'}) \) iff \( I \in G_1, I' \in G_2, N_I = N_{I'}, \text{ and } R_I = R_{I'} \).

(iv) **Update of location of external ED.** The URI of external entity \( J = (N_J, U_J, P_J) \) is updated to the URI of the external entity \( J' = (N_{J'}, U_{J'}, P_{J'}) \) iff \( J \in G_1, J' \in G_2, N_J = N_{J'}, \text{ and } U_J 
eq U_{J'} \).

(v) **Update of content notation of external ED.** The content notation of external entity \( J = (N_J, U_J, P_J) \) is updated to the content notation of the external entity \( J' = (N_{J'}, U_{J'}, P_{J'}) \) iff \( J \in G_1, J' \in G_2, N_J = N_{J'}, \text{ and } P_J 
eq P_{J'} \).

Note that if an entity \( g \) is changed from being an internal entity to being an external entity, or vice versa, then we consider as a pair of a deletion of an entity and an insertion of an entity.

### 8.2.4 DTD-Diff Algorithm

In this section, we present the DTD-DIFF algorithm. The outline of the algorithm is depicted in Figure 8.15(a). It takes as input two DTDs \( D_1 = (E_1, A_1, G_1) \) and \( D_2 = (E_2, A_2, G_2) \) representing old and new versions of a DTD, respectively, and returns an edit script \( Z \) containing the differences between \( D_1 \) and \( D_2 \). The algorithm consists of six phases: the parsing and hashing phase, the matching phase, the move detection phase, the attribute declaration changes detection phase, the entity declaration changes detection phase, and the edit script generation phase. We shall discuss each phase in turns.

**The Parsing and Hashing Phase**

Given two DTDs, \( D_1 \) and \( D_2 \), DTD-DIFF parses \( D_1 \) and \( D_2 \) into \( (T_{E_1}, A_1, G_1) \) and \( (T_{E_2}, A_2, G_2) \),
respectively, and computes their hash values. Note that \( T_1 \) and \( T_2 \) are two sets of content trees of \( E_1 \) and \( E_2 \), respectively. Since a content tree of an element type declaration has both ordered and unordered parts (the child nodes of the sequence and choice groups, respectively), the algorithm for computing the hash values must be able to address this issue. Given a node \( x \) in \( T_E \in T_{E_i} \) where \( i \in \{1, 2\} \), the hash value of node \( x \) is calculated as follows.

- If node \( x \) is a leaf node, then \( \text{Hash}(x) = \text{MD5Value}(\text{label}(x) \cdot \text{cardinality}(x)) \).
- If node \( x \) is a non-leaf node and a sequence group, then \( \text{Hash}(x) = \text{MD5-Value}(\text{Hash}(c_1) \cdot \text{Hash}(c_2) \cdot ... \cdot \text{Hash}(c_n) \cdot \text{label}(x) \cdot \text{cardinality}(x)) \).
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

Input: Two root nodes rl and r2
Output: A set of matching pairs M

1. M = empty set
2. push pair {rl, r2} into queue Q
3. WHILE (Q is not empty)
4. pop a pair (rl, r2) from queue Q
5. N » K U |tl, r2|
6. IF HashValue(rl) ≠ HashValue(r2) AND
   N1, N2 are non-leaf nodes THEN
7. FOR EACH childl IN children of rl
8. FOR EACH child2 IN children of r2
9. IF label(childl) = label(child2) THEN
10. ComputeCost(childl, child2)
11. ELSE
12. Cost(childl, child2) = 0
13. END IF
14. END FOR
15. matched_pairs = set of pairs resulting from
   minimum-cost bipartite matching among
   child nodes of rl and r2
16. FOR EACH pair(x, y) IN matched_pairs
17. push pair (x, y) into queue Q
18. END FOR
19. END IF
20. END WHILE
21. RETURN M

Figure 8.17: The Matching Algorithm.

where Hash(c_i) is the hash value of the child node of node x.

- If node x is a non-leaf node and a choice group, then Hash(x) = MD5-Value(Hash(c_1))
  - Hash(c_2) • ... • Hash(c_n) • label(x) • cardinality(x)), where Hash(c_i) is the hash value of the child node of node x, and Hash(c_1) < Hash(c_2) < ... < Hash(c_n).

Note that “•” denotes concatenation of strings. Function MD5Value is a hash function based on the MD5 Message-Digest algorithm [Riv]. We acknowledge that there is a very few probability of collisions of some hash functions (including MD5 hash algorithm) [BCJ05, Kli05, Kli03, WY05] that will influence the result quality of our approach. The hash function in DTD-DIFF can be replaced by other hash algorithms without any significant changes to the algorithm.

The CalculateHashValue algorithm is depicted in Figure 8.15(b). We also calculate the hash values of AD in A and ED in G. The hash value of AD A ∈ A is calculated as follows.

Hash(A) = MD5-Value(Hash(N_A) • Hash(s_1) • ... • Hash(s_x), where Hash(s_x) = MD5-Value(Hash(N_s) • Hash(Y_s) • Hash(D_s)), s_x ∈ S_A, and Hash(s_1) < Hash(s_2) < ... < Hash(s_x). The hash value of ED E ∈ G is calculated as follows. Hash(E) = MD5-Value(Hash(N_E) • H), where if E is an internal entity declaration, then H = Hash(R_E).

Otherwise, E is an external entity declaration, and H = Hash(U_E) • Hash(P_E). The overall complexity of calculating the hash values is O(∑_{i=1}^{T_1}(|T_{E_1}| × d_i) + ∑_{j=1}^{T_2}(|T_{E_2}| × d_j) + |A_1| + |A_2| + |G_1| + |G_2|) where |T_1| and |T_2| are the numbers of content trees in T_1 and T_2, respectively, |T_{E_1}| is the number of nodes in T_{E_1}, and d_i is the average out-degree of T_{E_1}.

The Matching Phase
Given two content trees of ETDs E_1 and E_2, denoted as T_{E_1} and T_{E_2}, respectively, DTD-DIFF invokes the Matching algorithm as depicted in Figure 8.17. The Matching algorithm
returns a set of matching pairs $M_{\text{min}}$. The principle behind the Matching algorithm in DTD-DIFF is based on the one in X-Diff [WDyC03]. That is, our matching technique finds the minimum-cost bipartite matchings of two content trees. However, there are critical differences between the Matching algorithm in DTD-DIFF and the one in X-Diff as we exploit the unique structural and semantic features of a DTD.

First, the Matching algorithm in X-Diff is invoked once. DTD-DIFF invokes the Matching algorithm as many as the number of ETDs. Observe that each ETD in a DTD has a unique name and hierarchy. Each root node in the content tree appears only once and mapping occurs only between nodes with the same signature. So each smaller content tree will be compared with another smaller tree from the second version having the root node with same name. For example, the content tree rooted at node labeled school in Figure 8.14(a) will be compared with the content tree in Figure 8.14(b) whose root has the same label. Note that this computation is independent from the remaining content trees. Second, the ComputeCost algorithm in Figure 8.18 that is invoked by the Matching algorithm in DTD-DIFF to compute the cost matching between $r_1$ and $r_2$ considers the cardinality changes (line 3, Figure 8.18). Note that the Matching algorithm in X-Diff does not consider the cardinality changes as it deals with XML documents, not DTDs. Third, unlike X-Diff which is based on unordered trees, a content tree can have ordered and unordered subtrees. Hence, in order to ensure our matching technique works on ordered subtrees as well, we adopt the technique used in XyDiff [CAM02] to find the largest order preserving sequences among those matching pairs in sequence groups (lines 26-28, Figure 8.18).

We now analyze the complexity of the matching phase. Let $|T_{E1}|$ and $|T_{E2}|$ be the numbers of nodes in the content trees $T_{E1}$ and $T_{E2}$, respectively. The complexity of finding the minimum-cost bipartite matchings is $O(|T_{E1}| \times |T_{E2}| \times \max\{\deg(T_{E1}), \deg(T_{E2})\})$
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

Input: $A_1$ and $A_2$
Output: Attribute Matching $M$

```
1 \text{IF} \ a_2 \ \text{is already matched} \ \text{THEN} \ \text{CONTINUE}
2 \text{ELSE IF} \ \text{Hash}(a_1) = \text{Hash}(a_2) \ \text{THEN}
3 \quad \text{Mark} \ a_1 \ \text{and} \ a_2 \ \text{that}
4 \quad \text{they are already matched}
5 \text{BREAK}
6 \text{ELSE IF} \ a_1.\text{name} = a_2.\text{name} \ \text{THEN}
7 \quad \text{IF} \ a_1.\text{type} <> a_2.\text{type} \ \text{THEN}
8 \quad \quad \text{Mark} \ a_1 \ \text{and} \ a_2 \ \text{that}
9 \quad \quad \text{their attribute types are updated}
10 \quad \text{END IF}
11 \quad \text{IF} \ a_1.\text{defval} <> a_2.\text{defval} \ \text{THEN}
12 \quad \quad \text{Mark} \ a_1 \ \text{and} \ a_2 \ \text{that}
13 \quad \quad \text{their default values are updated}
14 \quad \text{END IF}
15 \quad \text{END IF}
16 \text{M} = \text{M} \cup (a_1,a_2)
17 \text{BREAK}
18 \text{END IF}
19 \text{END FOR}
20 \text{END FOR}
21 \text{RETURN} \ M
```

Figure 8.19: The \textit{detectAttributeChanges} Algorithm.

$\log(\max(\{\text{deg}(T_{E1}), \text{deg}(T_{E2})\}))$, where $\text{deg}(T_{E1})$ and $\text{deg}(T_{E2})$ are the maximum out-degree in $T_{E1}$ and $T_{E2}$, respectively [WDyC03]. Suppose the numbers of ETDs in $D_1$ and $D_2$ are $\alpha_1$ and $\alpha_2$, respectively. Then, the total complexity of finding the minimum-cost bipartite matching can be estimated as $O(\min\{\alpha_1, \alpha_2\} \times |T_{E1}| \times |T_{E2}| \times \max\{\bar{d}_1, \bar{d}_2\} \times \log(\max\{\bar{d}_1, \bar{d}_2\}))$, where $|T_{E1}|$ and $|T_{E2}|$ are the average numbers of nodes of the content trees in $T_{E1}$ and $T_{E2}$, respectively, and $\bar{d}_1$ and $\bar{d}_2$ are the average out-degree of the content trees in $T_{E1}$ and $T_{E2}$, respectively.

The Move Detection Phase

After we have a set of matching pairs $M_{\text{min}}$, DTD-DIFF detects move operations. Formally, the move operation is defined as follows. Let $n_1$ and $n_2$ be two nodes in $T_{E1}$ and $T_{E2}$, respectively. Let $\text{parent}(n)$ be the parent node of node $n$. Then, node $n_1$ is moved to be node $n_2$ iff $(\text{parent}(n_1), \text{parent}(n_2)) \notin M_{\text{min}} \text{ and } \text{Hash}(n_1) = \text{Hash}(n_2)$. Note that we only consider a move operation if the hash values of $n_1$ and $n_2$ are the same. This is because if the hash values of $n_1$ and $n_2$ are different, then we need to check the differences in the subtrees rooted at $n_1$ and $n_2$. If the hash values of $n_1$ and $n_2$ are different, then the algorithm detects it as a deletion of $n_1$ and an insertion of $n_2$.

Now, we discuss how the move operations are detected. Let $P$ and $Q$ be two lists of the subtrees from the first and second versions, respectively, that have no matching subtrees in $M_{\text{min}}$. Subtrees in $P$ and $Q$ are sorted by their size in decreasing order. For each subtree in $P$, the algorithm checks whether there is a subtree in $Q$ that have the same hash value. If $p_i \in P$ and $q_j \in Q$ have the same hash value, then the algorithm marks that subtree $p_i$ in the first version is moved to be subtree $q_j$ in the second version. The complexity of the
algorithm for finding move operation is \( O(n \times \log(n)) \), where \( n \) is the number of nodes in the content tree.

**The Attribute Declaration Change Detection Phase**

Recall that attribute list can be seen as a collection of attributes. Given two collections of attributes of an element in the first and second versions, the changes to the attribute list can be detected by using the algorithm in Figure 8.19. The complexity of the algorithm for finding the changes on the attribute lists is \( O(n \times \log(n)) \), where \( n \) is the number of attributes defined in the DTD. Note that we do not consider the update of the attribute name for the reasons discussed in the preceding section. We consider the update of the attribute name is represented as a pair of deletion and insertion of an attribute.

**The Entity Declaration Change Detection Phase**

The change detection mechanism of EDs is quite straightforward and similar to the approach for detecting changes to attribute declarations. Hence, we do not elaborate on this step further. The complexity of the algorithm for finding the changes on the entity declarations is \( O(n \times \log(n)) \), where \( n \) is the number of entity declarations defined in the DTD.

**The Edit Scripts Generation Phase**

The edit script \( Z \) is generated as follows. (1) An edit script \( Z \) is initialized as a set of move operations detected in the preceding step. (2) Then, for all unmatched nodes in
CHAPTER 8. DETECTING THE CHANGES TO HYBRID XML

the first tree, delete operations are added into edit script Z. (3) Next, for all unmatching nodes in the second tree, insert operations are added into edit script Z. (4) For all pairs of matching nodes that have different cardinality, cardinality update operations are added into edit script Z. (5) For all pairs of matching nodes that belong to sequence groups and have incorrect local order, local order move operations are added into edit script Z. (6) The changes to the attributes lists are added into edit script Z. (7) Finally, the changes to the entity declarations are added into edit script Z. The overall complexity of this step is $O(\sum_{i=1}^{T_1}(|T_{E1}|) + \sum_{j=1}^{T_2}(|T_{E2}|) + |A_1| + |A_2| + |G_1| + |G_2|)$.

8.2.5 Experimental Results

We have implemented DTD-DIFF entirely in Java. The experiments were conducted on a Microsoft Windows XP Professional machine having Pentium 4 1.7 GHz processor with 512 MB of memory. We use both real world DTDs and a set of synthetic DTDs generated by using our DTD generator. The second versions of DTDs are generated by using our DTD changes generator. We vary the numbers of element types, the percentage of changes, the out-degree of each element types, and the depth of each element types.

We compare the performance of DTD-DIFF with the C version of XyDiff [CAM02] (downloaded from http://pauillac.inria.fr/cdrom/www/xydiff/index-eng.htm)$^1$ and the Java version of X-Diff [WDyC03] (downloaded from http://www.cs.wisc.edu/~yuanwang/xdiff.html). Further, as X-Diff and XyDiff are not designed for detecting the changes on DTDs, we convert the DTDs into XML Schema (XSD) [Sch] using Syntex dtd2xs tool [Syn] before detecting the changes. For example, given a DTD depicted in Figure 8.11(a), the equivalent XSD generated by using Syntex dtd2xs is depicted in Figure 8.12(a) (partial view only). Note that we have also investigated several other DTD-to-XML Schema conversion tools that are freely available publicly. For instance, we used LuMrix dtd2xs [IE] to convert DTDs into XSDs. To the best of our knowledge, these tools produced almost similar XSD files and the differences did not significantly influence the performance of X-Diff and XyDiff. Note that the results of X-Diff and XyDiff suffer from the limitations discussed in the preceding section.

Execution Time vs Number of Element Types

We first study the performance of DTD-DIFF by varying the number of element types. We

---

$^1$Unfortunately, despite our best efforts (including contacting the authors), we could not get the Java version of XyDiff.
set the out-degree and depth of each element type to “5” and “3”, respectively. Note that
the average of the maximum depth of real DTDs is “3” [Cho02]. The number of attributes
of each element is set to “3”. We set the percentages of changes to “1%” and “9%”. The
characteristic of the data sets used in this set of experiments is depicted in Figure 8.20(a).
Figure 8.21(a) shows the execution time of converting DTD into XSD file using Syntex
dtd2xs.

Figure 8.21(b) depicts the performance of DTD-Diff, X-Diff, and XyDiff when the
percentage of changes is set to “1%”. We observed that DTD-Diff significantly outperforms
X-Diff and XyDiff. DTD-Diff is 8.6–155 times faster than X-Diff and 7.63–26.42 times
faster than XyDiff. Figure 8.21(c) presents the performances of DTD-Diff and X-Diff
when the percentage of changes is set to “9%”. In this case, DTD-Diff is 5–272 times
faster than X-Diff and 4.68–20.10 times faster than XyDiff. X-Diff failed to detect the
changes when the numbers of elements are more than or equal to 250 due to lack of main
memory. The inability of X-Diff to process large number of nodes in XML data is also
highlighted in [LB05b, LBM05].

Figure 8.21: Experimental Results (1).
We now elaborate on why our approach significantly outperforms X-Diff. First, the tree representations of XSD files (XSD tree) contain elements with same names. On the other hand, in DTD-DIFF, each root node of the content trees in a DTD has a unique name. As a result, there exists a one-to-one mapping between a content tree in the old version to another content tree in the new version. Consequently, X-Diff does more number of bipartite matching compared to DTD-DIFF. Second, the number of nodes in the content trees is lesser in most cases compared to an XSD tree. This further reduces the number and cost of bipartite matching in DTD-DIFF. To elaborate further, let $|T_1|$ and $|T_2|$ be the numbers of nodes in the XSD trees of DTDs $D_1$ and $D_2$, respectively. The complexity of finding the minimum-cost bipartite matchings between $|T_1|$ and $|T_2|$ in X-Diff is $O(|T_1| \times |T_2| \times \max\{\deg(T_1), \deg(T_2)\} \times \log(\max\{\deg(T_1), \deg(T_2)\}))$ [WDyC03]. The improvement of DTD-DIFF over X-Diff can be estimated as $O(|T_1| \times |T_2| \times \max\{\deg(T_1), \deg(T_2)\} \times \log(\max\{\deg(T_1), \deg(T_2)\})) / O(\min\{\alpha_1, \alpha_2\} \times |T_1| \times |T_2| \times \max\{d_1, d_2\} \times \log(\max\{d_1, d_2\})))$. Assuming that $|T_1| = |T_2| = n$, $|T_1| = |T_2| = t$, and $\max(\deg(T_1), \deg(T_2)) = x$, the complexity comparison becomes $O(t^2 / \alpha n^2) = O(t^2 / \alpha n^2)$. Note that $|T_1|$ and $|T_2|$ include the attribute and entity declarations of $D_1$ and $D_2$. However, in the case of DTD-DIFF $|T_1|$ and $|T_2|$ do not include these declarations as their changes are detected separately. Based on our discussion in the preceding section, it is less expensive in DTD-DIFF to compute changes to EDs and ADs. Furthermore, numbers of nodes in the XSD files are larger than the number of nodes in the content trees (from 2.8 up to 5.8 times larger, Figure 8.20). Therefore, $\alpha \times n^2 \leq t^2$. Hence, the performance of DTD-DIFF is always faster or in the worst case comparable to X-Diff. DTD-DIFF is faster than XyDiff due to the similar reasons as above.

We also study the performance of DTD-DIFF and X-Diff for detecting the changes to the real world DTDs [DRG, XML]. Figure 8.22(a) depicts the characteristics of the real world DTDs. We set the percentage of changes to 3%. Figure 8.22(b) depicts the performances of DTD-DIFF and X-Diff. We notice that X-Diff has slightly better performance than DTD-DIFF. This is primarily due to the characteristics of the data. For instance, although NewsML1.1 has 117 elements, the performance of DTD-DIFF is comparable to X-Diff! Observe that for synthetic data set with similar size, DTD-DIFF outperforms X-Diff significantly. This is because in NewsML1.1, only 6 out of 117 ETDs have nested content and the maximum depth of NewsML1.1 DTD is only 2. Hence, cost of bipartite matching
is almost the same. In summary, X-Diff performs relatively better than DTD-DIFF when the DTDs have simple and “flat” structure. When the DTD structure is complex, DTD-DIFF outperforms X-Diff as shown using synthetic dataset. Also, note that DTD-DIFF is still better than X-Diff because of the inaccuracies and incompleteness in the results generated by X-Diff due to the limitations highlighted in Section 1. Compared to XyDiff, we observe that DTD-DIFF is up to 12.61 times faster.

**Execution Time vs Percentage of Changes**

In this set of experiments, we study the effects of the percentages of changes on the execution time of DTD-DIFF, X-Diff, and XyDiff. We use the E005-B05-D02 data set, whose number of element types, out-degree, and depth are 5, 5, and 2, respectively, as the first version of the DTD. We vary the percentages of changes from “1%” to “20%”. Figure 8.21(d) depicts the execution time of DTD-DIFF, X-Diff, and XyDiff for different percentages of changes. We observe that the percentage of changes slightly affect the performances of DTD-DIFF, X-Diff, and XyDiff.

**Execution Time vs Out Degree**

In this set of experiments, we study the effects of the number of out-degree of each element type on the execution time of DTD-DIFF, X-Diff, and XyDiff. We set the number of element types and the depth to “25” and “2”, respectively. We set the percentages of changes to “1%” and “9%”. We vary the out-degree of each element type from “5” to “50”. The characteristic of the data sets used in this set of experiments is depicted in Figure 8.20(b). Figures 8.23(a) and 8.23(b) depict the performance of DTD-DIFF, X-Diff, and XyDiff for
different numbers of out-degree of each element type when the percentages of changes are set to "1%" and "3%", respectively. We observe that DTD-DIFF is up to 325 times faster than X-Diff. DTD-DIFF is 2.52-15.48 times faster than XyDiff. This is because of the reasons discussed above. We also notice that X-Diff cannot detect the changes to XSD files when the out-degree is more than or equal to 25 due to the lack of main memory.

**Execution Time vs Depth**

In this set of experiments, we study the effects of the depth of content tree on the execution time of DTD-DIFF, X-Diff, and XyDiff. We set the number of element types and the out-degree to “25” and “5”, respectively. We set the percentages of changes to “1%” and “9%”. We vary the out-degree of each element type from “1” to “8”. The characteristic of the data sets used in this set of experiments is depicted in Figure 8.20(c). Figures 8.23(c) and 8.23(d) depict the performance of DTD-DIFF, X-Diff, and XyDiff for different depth of each content tree when the percentages of changes are set to “1%” and “9%”, respectively. We observe that DTD-DIFF is up to 89 times faster than X-Diff and 9.5-38 times faster than...
XyDiff. X-Diff failed to detect the changes when the depth is more than or equal to 8 due to the lack of main memory.

**Result Quality**

We also examine the quality of deltas detected by DTD-DIFF. We use E010-B05-D02 data set and the percentages of changes are varied between “1%” to “10%”. The second versions are generated by using our DTD change generator. Then, we calculate the result quality, that is, the ratio between the number of edit operations detected by DTD-DIFF and the optimal one. Figure 8.24 depicts the ratios. We observe that DTD-DIFF is able to detect the optimal deltas until percentage of changes is set to “5%”. Afterwards, DTD-DIFF detects almost optimal deltas. This is because, in some cases, a move operation is detected as a pair of deletion and insertion. Note that we do not compare the result quality of DTD-DIFF to other approaches as, to the best of our knowledge, DTD-DIFF is the first approach for detecting the changes to DTDs. We do not compare the result quality of DTD-DIFF to the one of X-Diff (when we use XSD files) as the types of changes of DTD and XML are different.

**8.3 Summary**

In this chapter, we described two approaches for detecting the changes to hybrid XML documents (called XANDY−H) and DTD (called DTD-DIFF). We presented the algorithm as well as the SQL queries used in XANDY−H to determine the best matching subtrees (Phase 1). We elaborated on how XANDY−H detects the changes (Phase 2). We also presented the performance study of XANDY−H. The performance studies of XANDY−H and DTD-DIFF are also presented.
Chapter 9

Conclusions and Future Work

The eXtensible Markup Language (XML) has recently emerged as a new standard for representing and exchanging data on the Internet. More and more applications have adopted XML for exchanging information on the Web. XML document will also be changed when the information on the Web is changed. Consequently, being able to quickly detect the changes in XML documents is an important problem. The objective of this dissertation is to investigate relational-based approaches for detecting the changes to XML documents.

In this chapter we first highlight the contributions made in the dissertation. Next, we present the conclusions of our implementations and experiments. Finally, based on our findings possible directions for future work are highlighted.

9.1 Contributions

We have extensively investigated the use of relational database systems for detecting the changes to XML documents. To that end, the following contributions have been made.

- We have shown that it is indeed possible to utilize relational databases for detecting the changes. We are able to use the schema-conscious and schema-oblivious approaches as the underlying relational schema for storing XML documents before we detect the changes to these XML documents.

- We have designed and implemented a relational-based approach called XANDY that uses schema-oblivious approach, namely, SUCXENT schema for storing XML documents. XANDY has three variants for detecting the changes to unordered XML documents (called XANDY-U), ordered XML documents (called XANDY-O), and hybrid XML documents (called XANDY-H).
• We have presented HELIOS that uses schema-conscious approach, namely, Shared-Inlining schema for storing XML documents. Recall 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. HELIOS has two variants for detecting the changes to unordered XML documents (called HELIOS-U) and ordered XML documents (called HELIOS-O).

• We proposed a new type of XML documents, namely, hybrid XML based on our analysis of different real world XML documents that they may not always be purely ordered or purely unordered.

• We have implemented an approach for detecting the changes to DTD called DTD-DIFF. To the best of our knowledge, DTD-DIFF is the first approach that addresses the problem of detecting the changes to DTDs. Note that DTD can be represented as a set of hybrid trees. This is because DTDs often contain sequence (denoted by "",")) and choice (denoted by "|")) groups. The order of elements in a sequence group is important, while the order of elements in a choice group is not significant.

• We studied the performance of XANDY and HELIOS and compared it with existing approaches. We also have analyzed the result quality of our approaches and compared it to the one of state-of-the-arts approaches.

9.2 Conclusions

We carried out several experiments based on the contributions discussed in the previous section. The key findings of these studies are as follows.

• The relational-based approaches are more scalable than the memory-based approaches. This is because the memory-based approaches require both XML tree to be memory resident. This problem is exacerbated by the fact that these trees are typically much larger (about 5-10 times) than their XML documents. The relational-based approaches depend on the secondary storage, in our case, hard drive.
CHAPTER 9. CONCLUSIONS AND FUTURE WORK

- For small data sets, the memory-based approaches are faster than the relational-based approaches. This is due to the I/O cost of accessing data stored in the relational databases.

- HELIOS performs better than XANDY. This is due to the fragmented nature of the data stored in a schema-conscious approach.

- The performance of HELIOS and XANDY is affected by the following factors.
  - The choice of relational database systems will affect the performance of our approaches. This is because each relational database system has its own optimizer that generates the query execution plans. In many cases, the query execution plans generated by an optimizer are different from the ones of another optimizer.
  - Number of nodes in XML documents significantly influences the performance of our approaches.
  - The file size of XML documents has no significant impacts to the execution time of our approaches. This is because file size does not have effects to the number of tuples in relational database.
  - Our approaches are affected by the percentage of changes.
  - The depth of XML documents also affects our approaches. This is because our approaches find the best matching subtrees in the bottom-up fashion.
  - The similarity score threshold \( \theta \) does not significantly influence the performance of HELIOS and XANDY.

- HELIOS and XANDY are able to produce better result quality than the existing approaches. HELIOS-O and XANDY-O have better result quality then XyDiff. Similarly, HELIOS-U and XANDY-U have better result quality then X-Diff.

- The result quality of our approaches is influenced by the similarity score threshold \( \theta \), the percentage of changes, and the distribution of changes.

- DTD-DIFF has superior performance compared to X-Diff and XyDiff when we use them for detecting the changes to XSD files. In addition, the deltas detected by DTD-DIFF consist of DTD types of changes.
9.3 Future Works

Based on our findings we believe that the following present promising directions for future work.

- Optimization of Phase 1. In our approaches, Phase 1 ("Finding best matching subtrees") is the bottleneck. The worst case is that Phase 1 can contribute up to 99% of the overall performance of our approaches. Therefore, optimization of Phase 1 is necessary in order to speed up the performance of HELIOS and XANDY.

- Extending HELIOS. Currently, HELIOS does not support XML documents that have recursive schema. In the real-world, we can find many XML documents that have recursive schema. Hence, the extensions will generalize the framework in HELIOS.

- Studying the effects of various relational schemas for storing XML documents. Many relational schemas for storing XML documents [FK99, JLWY02a, JLWY02b, TVB+02, YASU01, RFHR03, BFRS02, PBM04a, BWLM03] have been proposed. Hence, the investigation on using various relational schemas for relational-based approaches is interesting.

- XML schema change detection. Beside DTD, XML schema is also used to define legal structure of XML documents. Basically, XML schema is in XML format; however, we are not able to use XML change detection algorithm directly as it may generate semantically incorrect deltas. We need to exploit the semantics of XML schema to guide the change detection process.

- Tree Matching Problem. Phase 1 in our approaches can be extended to address the tree matching problem. In other words, we are able to investigate the relational-based approach that addresses the tree matching problem.
References


REFERENCES


[BLL+02] Stéphane Bressan, Mong Li Lee, Ying Guang Li, Zoé Lacroix, and Ullas Nambiar. The XO07 Benchmark. In *Proceedings of the first VLDB Workshop on Efficiency and Effectiveness of XML Tools, and Techniques (EEXTT)*, Hong Kong, China, August 2002.


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES

on Information and Knowledge Management (ACM CIKM 2005), Bremen, Germany, October-November 2005.


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


Index

Best matching subtrees, 44, 48

Helios-O, 62
  detecting the changes phase, 104
  finding best matching subtrees phase, 78

Helios-U
  detecting the changes phase, 168
  finding best matching subtrees phase, 161

Inlining approach, 54

Internal node
  deletion, 48
  insertion, 48

Leaf node
  deletion, 49
  insertion, 49

Matching leaf node groups, 45

Matching leaf nodes, 44
  fixed, 44
  shifted, 44

Matching sibling orders, 45

Matching subtrees, 45

Minimum XDelta, 42

Move operation, 52

Similarity score, 46

Sucxent, 52

Types of changes, 48

Update operation, 51

Xandy-O
  bottom-up approach, 61, 70
  detecting the changes phase, 88
  finding best matching subtrees phase, 63
  top-down approach, 60, 63

Xandy-U
  detecting the changes phase, 165
  finding best matching subtrees phase, 155

XML
  hybrid, 3, 11
  ordered, 3, 4
  unordered, 3, 8

249
Appendix A

List of Publications

During the four-year PhD candidature, the author has contributed a total of 10 international publications.

**Refereed Journal Paper**


**Refereed Conference Papers**


- **ERWIN LEONARDI and SOURAV S BHOWMICK.** OXONE: A Scalable Solution for Detecting Superior Quality Deltas on Ordered Large XML Documents. *In the Proceedings of the 25th International Conference on Conceptual Modelling (ER 2006)*. Springer Verlag, Tuscon, USA, Nov, 2006. (Acceptance rate 23%)
APPENDIX A. LIST OF PUBLICATIONS

- **ERWIN LEONARDI, TRAN T HOAI, SOURAV S BHOWMICK,** and **SANJAY MADRIA.**


- **ERWIN LEONARDI, SOURAV S BHOWMICK,** and **SANJAY MADRIA.** XANDY: Detecting Changes on Large Unordered XML Documents Using Relational Databases. *In the Proceedings of the 10th International Conference on Database System for Advanced Applications* (DASFAA 2005). Springer Verlag, Beijing, China, Apr, 2005. (Acceptance rate 22%)
