Schema-Less, Semantics-Based Change Detection for XML Documents

Shuohao Zhang[1], Curtis Dyreson[1], and Richard T. Snodgrass[2]

[1] Washington State University, Pullman, Washington, U.S.A.
{szhang2, cdyreson}@eecs.wsu.edu

[2] The University of Arizona, Tucson, Arizona, U.S.A.
rts@cs.arizona.edu

Abstract. Schema-less change detection is the processes of comparing successive versions of an XML document or data collection to determine which portions are the same and which have changed, without using a schema. Change detection can be used to reduce space in an historical data collection and to support temporal queries. Most previous research has focused on detecting structural changes between document versions. But techniques that depend on structure break down when the structural change is significant. This paper develops an algorithm for detecting change based on the semantics, rather than on the structure, of a document. The algorithm is based on the observation that information that *identifies* an element is often conserved across changes to a document. The algorithm first isolates identifiers for elements. It then uses these identifiers to associate elements in successive versions.

1 Introduction

Change detection is a process that identifies changes between successive versions of a document or data collection. At the logical level, change detection aids in understanding the temporal behavior of data. Putting data into the context of its evolution usually entails more meaningful information. This is particularly true of data on the web, which often has a high rate of change [2,6]. At the physical level, change detection helps an archival system reduce space by storing only the changes to a new version because the size of the change is generally a small fraction of the entire version. In systems where storage is not a concern but data is shipped across a network, change detection can reduce network traffic, since often only the changes, and not the entire document, can be transferred.

Change detection is also significant in *temporal query support* and *incremental query evaluation*. In contrast to queries that are issued against a single version of the data, temporal queries can involve both current and past versions [11,12]. Temporal queries that access more than one version are important in mining historical facts and predicting future trends. The semantics of many temporal queries depends on identifying which data has changed and which has continued to reside unchanged in a database. Incremental query evaluation, when applicable, can significantly reduce the cost

X. Zhou et al. (Eds.): WISE 2004, LNCS 3306, pp. 279–290, 2004.
© Springer-Verlag Berlin Heidelberg 2004

of query evaluation. Some *continuous* (and non-continuous) queries can be incrementally evaluated, just by using the change. A continuous query is re-evaluated when *new* data arrives, for instance, as the query reads from a data stream [15,16]. For certain types of queries it is sufficient to combine the result of the query on the changed data with the previous result, rather than re-evaluate the query on the entire database.

This paper is organized as follows. Section 2 presents an example that motivates this research. We show that when the structure of a document changes while the data remains the same, a new method is needed to associate nodes in successive versions of a document. Related work in structural change detection is presented in Section 3. Section 4 formalizes the notion of *semantic identifiers* for Extensible Markup Language (XML) documents and outlines an algorithm for computing identifiers. Next, the paper shows how to use the semantic identifiers to detect changes in XML by matching elements based on their identifiers. A match means that an element has retained its semantic identity across successive versions. The paper then concludes with a discussion of future work.

2 Motivation

Fig. 1 shows an XML document. Fig. 2 shows the next version of the document. The change from the old version to the new version can be effected in two steps: an update and an insertion. The underlined parts in Fig. 1 and Fig. 2 reflect the change. The rest of the new version is an *exact* copy of the old version.

But consider a different, more substantial change. Fig. 3 shows an alternative version of the document fragment in Fig. 1. It is an alternative version in the sense that it has basically the same information, just arranged in a different schema. What has changed from Fig. 1 to Fig. 3? The two fragments are far from identical. Intuitively, it requires a significant number of element insertions and deletions.

The high cost of change in this particular situation, however, is not our primary concern. Regardless of how large the change is, the cost is always an order of the size of the document. The problem lies in the fact that significant structural change makes it difficult to *associate* elements in different versions. An association is a correspondence between elements in successive versions. The association establishes that an element in one version has a successor in the next version of the document. The association is possibly a new version of the element. For instance, the Pocket Star <publisher> element in Fig. 3 should be associated with the Pocket Star <publisher> element in Fig. 1; ostensibly, it is a new version of that element. But each <publisher> is structurally very different and therefore cannot be associated by change detection processes based on recognizing structural copies. This creates an obstacle for temporal query or incremental query evaluation. If we are unable to appropriately associate elements in successive versions, these applications cannot be implemented correctly. The example sketched above suggests that if the structural change is significant, even though semantically the same data is present, it becomes difficult to associate elements.

```
<author>
  <name>Dan Brown</name>
  <book>
    <title>The Da Vinci Code
    </title>
    <publisher>Doubleday
    </publisher>
    <listprice>$24.95
    </listprice>
  </book>
  <book>
    <title>Angels & Demons
    </title>
    <publisher>Pocket Star
    </publisher>
    <listprice>$7.99</listprice>
  </book>
</author>
```

Fig. 1. The original version

```
<author>
  <name>Dan Brown</name>
  <book>
    <title>The Da Vinci Code
    </title>
    <publisher>Doubleday
    </publisher>
    <saleprice>$14.97</saleprice>
    <isbn>0385504209</isbn>
  </book>
  <book>
    <title>Angels & Demons
    </title>
    <publisher>Pocket Star
    </publisher>
    <listprice>$7.99</listprice>
  </book>
</author>
```

Fig. 2. A new version

```
<publisher>Doubleday
  <book>
    <title>The Da Vinci Code</title>
    <author>
      <name>Dan Brown</name>
    </author>
    <listprice>$24.95</listprice>
  </book>
</publisher>
<publisher>Pocket Star
  <book>
    <title>Angels & Demons</title>
    <author>
      <name>Dan Brown</name>
    </author>
    <listprice>$7.99</listprice>
  </book>
</publisher>
```

Fig. 3. An alternative new version

At issue is how to define "change." Previous research considered two documents, or portions thereof, the same if and only if they are *identical* (see Section 3 for a review of previous research). This requires not only textual equality but structural as well. Some researchers have considered unordered trees as the basis of comparing versions, which unlike the XML data model, ignores order among siblings.

This paper proposes a semantics-based change detection framework. Nodes that are *semantically equivalent* are considered unchanged and will be associated, regardless of their structural contexts. Finding all such associations between two versions has two important benefits: (1) it allows elements to exist across successive versions of a document (thus providing good support for temporal queries, for example), and (2) it detects semantic changes (which also includes "easier" changes, i.e., those that do not involve significant amounts of structural change).

3 Related Work

Early work in change detection mainly deals with string matching [1,14,18,19,23]. The subject is thus "flat" (plain-text) documents without hierarchical structures like those in SGML or XML. Plain-text change detection does not work well for XML because it is insensitive to hierarchical structure. It is also poorly-suited to finding semantic change, which is the goal of this paper. There has been some research in change detection for HTML [8] and general hierarchical structured documents [5,7]. To achieve efficiency, some simplifying assumptions are made. For example, [7] assumes that any leaf in one version has at most one leaf in another version "close" enough to be its match. This may not be true for such XML documents, as illustrated in Fig. 3.

In almost all research on XML, the data model is considered or presumed to be a tree. It is thus natural to make use of the results from tree matching (sometimes also called tree correction) [13,17,20] in XML change detection. It is important to point out that most research adopts an *ordered*, *labeled tree* data model [4]. The best known algorithm for general ordered labeled tree matching is by Zhang and Shasha [24]. To the best of our knowledge, only a few papers have considered the scenario where XML is modeled as an unordered labeled tree [22,26], in part because the tree matching problem for unordered labeled trees is NP-complete [25]. Due to space limitations, we omit presenting the complexities of the algorithms mentioned above since they all consider structural equality for a match while we focus on semantic equivalence.

Our semantic change detection technique is based on finding a (*semantic*) *identifier* for each node in an XML data model. An identifier is like a *key* for XML, but does not play a role in validation. Buneman et al. define a key as the pairing of a *target path* with a set of *key path expressions* [3]. Both the target path and key path expressions are *regular expression queries* (similar to XPath queries) on the data model for the document. Our work on identifiers differs from Buneman et al.'s research in that it is possible for the identifiers (but not keys) of two nodes to evaluate to the same value (when they have the same semantics) and we focus on a method to compute and use identifiers to associate nodes across multiple versions of a document's data model.

4 Semantic Identifiers

This section develops an algorithm for computing semantic identifiers. Basically, an identifier is a query expression that can be used for element identity, that is, to distinguish one element from another. Semantic identity is related to identity based on structure alone, but some elements can have exactly the same structure, yet be semantically different.

4.1 Structural Identity

In a given XML data model, each element and text node is of a specific *type*, which we will denote as *T*. The type is the concatenation of labels (element names) on the path from the root to a node, separated by a '/'. The type has been referred to in the literature as the *signature* [22]. A text node and its parent (an element node) will have the same type, but they are distinguished by the fact that they are different *kinds* of nodes. We will refer to the final label in the type as the *abbreviated type*. It suffices to use the abbreviated type except when two types happen to have the same abbreviation.

The structure of an element node is its type and the set of all its descendants (assuming lexical order is not important, a list of descendants otherwise). We will consider a pair of elements to be *structurally different* if the elements are of different

types or if there is some descendant of one element that is structurally different from every descendant of the other element. Otherwise, the elements are considered to be *structurally identical*.

4.2 Relating Semantic to Structural Identity

This section develops a notion of semantic identity as different from, but related to, structural identity. We give two axioms that capture the intuition with which we reason about semantic identity. Thus these axioms serve as a bridge that connects intuitive understanding to the rigor necessary for computer processing.

Axiom I: *Nodes that are structurally different are semantically different.*

Let's consider text nodes first, and then element nodes. The structure of a text node is its type and value. Axiom I states that two text nodes are considered to be different, semantically, when their structures are different. As an example, consider the document in Fig. 3. The text nodes corresponding to "Angels & Demons" and "The Da Vinci Code" are different in semantics because they are textually different. But the text nodes corresponding to "Dan Brown" are structurally identical, and therefore could be semantically identical. Each <book> element has <title>, <author> and <listprice> subelements. Since the two text children of *title* nodes are "The Da Vinci Code" and "Angels & Demons," the two <book> nodes are semantically different regardless of the <author> or <listprice> nodes.

If two element nodes are semantically different, it is not necessarily the case that they are structurally different. In fact, it is possible for two element nodes to be structurally identical but semantically different. This is stated in the following axiom.

Axiom II: *Nodes that are structurally identical are semantically identical if and only if their respective parents are semantically identical, or if they are both root nodes.*

Axiom II states that nodes that have the same structure also have the same semantics if and only if their parents are semantically the same. Axiom I distinguishes nodes that have different content; but when nodes have exactly the same content, Axiom II offers an alternative to distinguish them by their *context*. The context is the semantic identity of the parent node. For example, the two <name> nodes in the data model for the XML document in Fig. 3 both have a text child "Dan Brown" and are thus structurally equivalent. Are they semantically equivalent? It depends on their context. If we inspect their parents' semantics, we find that the two <author> nodes are structurally different (in the <book> subelement, or similarly, in <listprice>), and so by Axiom I are semantically different. Therefore the two <name> nodes are structurally identical but semantically different since each is in the context of a different book.

If two structurally equivalent nodes have semantically identical parents, then they are regarded as identical. This is reasonable because we cannot semantically distinguish two exact copies when they are enclosed in the same context.

4.3 Identifiers

This section defines several terms that are important to the algorithm for semantics-based change detection presented in the next section.

An identifier is based on the evaluation of XPath expressions [21] so we first define what it means to evaluate an XPath expression.

Definition [XPath evaluation]. Let *Eval*(n, E) denote the result of evaluating an XPath expression E from a context node n. Given a list of XPath expressions, $L = (E_1, \ldots, E_k)$, then $Eval(n, L) = (Eval(n, E_1), \ldots, Eval(n, E_k))$. ∎

Since an XPath expression evaluates to a list of nodes, *Eval*(n, L) evaluates to a list of lists.

Definition [Identifier]. An *identifier* for a type, T, is a list of XPath expressions, L, such that for any pair of type T nodes, x and y, x and y are semantically different if and only if $Eval(x, L) \neq Eval(y, L)$. ∎

An identifier serves to distinguish nodes of the same type. Two nodes are considered semantically the same if and only if their identifiers evaluate to the same result. Two lists are considered equivalent if they have the same cardinality and are equal at each position.

The choice of XPath expressions as the basis for specifying an identifier is a means rather than an end. It could be any mechanism that is able to properly locate nodes in a data model. We use XPath since it is widely adopted and supported.

Definition [Identifier map]. An *identifier map* (denoted M) is a relation that maps each type to its corresponding identifier (an XPath expression list), i.e.,

$$M = \{(T, L) \mid L \text{ is an identifier for type } T\}.$$ ∎

Identifiers are constructed from XPath expressions that locate values (text nodes) in the subtree rooted at a node.

Definition [Type-to-leaf path list]. The *type-to-leaf path list* for a type, T, denoted *typeL*(T), is a list of XPath expressions such that *typeL*(T) is a sorted list of XPath expressions, **sort**(S), where

- $E = \{e \mid e \text{ is of type } T\}$ is a set of all of the elements of type T in a document,
- $D = \{d \mid d \text{ is a text descendant of some } e \in E\}$ is a set of all of the text descendants of type T elements (if T is a text type then self() is the only descendant), and
- $S = \{s/\text{text()} \mid s = \textbf{suffix}(T, T) \text{ where } T \text{ is the type of some } d \in D\}$ is a set of all of the relative XPath expressions that locate a text descendant from a context node of a type T element. (Note that if T and T are the same then S includes text().) ∎

A type-to-leaf path list is a specific list of XPath expressions that locate text values that are descendants of nodes of a particular element type. In a given XML document, each type T has exactly one *typeL*(T). In the document shown in Fig. 1, for example, *typeL*(*author/book*) = (title/text(), publisher/text(), listprice/text()).

Note that for the document shown in Fig. 1, *typeL*(*book*) should contain one more XPath expression: text(). We believe that trivial text nodes, i.e., text nodes whose contents are all white-space characters, are insignificant in semantics. Thus the expression text() appears in a type-to-leaf path list *typeL*(T) only if there exists at least one type T node with a non-trivial text child.

Definition [unique with respect to *typeL*(T)]. Suppose that node n is of type T. Then n is *unique with respect to typeL*(T), if and only if *typeL*(T) is an identifier for type T. That is, if and only if for any n' of type T,

$$Eval(n, typeL(T)) \neq Eval(n', typeL(T)).$$ ∎

4.4 Computing Identifiers

The algorithm to compute identifiers will operate in a bottom-up fashion, working from the leaves towards the root of the tree. The following definitions describe positions in the bottom-up traversal. We use the term "floor" to evoke the idea that the algorithm ascends the tree like a person might climb floors in a building.

Definition [Floor-0 node]. All text nodes and only text nodes are *floor-0 nodes*. ∎

Definition [Floor-*k* node]. A node is a *floor-k node* if and only if the maximal floor among its children is k-1. ∎

Note that not all nodes of a certain type are of the same floor, and not all nodes of the same floor are of the same type. Types are computed top-down while floors bottom-up in the data model tree. Both concepts are important in computing identifiers.

Definition [local identifier]. An identifier is a *local identifier* if the XPath expressions evaluate to descendants of the context node; otherwise it is non-local. ∎

An identifier contains XPath expressions that evaluate to some leaf nodes in the document tree. It is either a local identifier or a non-local identifier. A non-local identifier locates at least one leaf node that is not a descendant of the context node. For example, the <name> node in Fig. 1 is identified by its text content, Dan Brown; so (text()) is a local identifier for <name>. On the other hand, the two <name> nodes in Fig. 3 have identical contents. It is impossible for the identifier of this type to contain only descendants of the <name> nodes. Thus <name>'s identifier must be non-local.

The algorithm for computing identifiers is shown in Fig. 4. The algorithm consists of two phases. Phase 1 finds all local identifiers, working bottom-up from the floor-0 nodes. This phase corresponds to Axiom I. When Phase 1 terminates, all semantically distinct nodes that Axiom I can determine are found. Phase 2 recursively computes the identifiers for the remaining types. This corresponds to Axiom II. When Phase 2 terminates, all semantically distinct nodes are found. Any remaining node is a redundant copy of another node in the document.

The total cost of the algorithm is bounded by $O(n*\log(n))$ where n is size of the document tree.

Pre: $M = \{(T_k, ())\}$ ($1 \le k \le$ number of different types)

Post: $M = \{(T_k, L_k) \mid L_k$ is a identifier for type $T_k\}$

Phase 1: find local identifiers

1) $i = 0$;
2) For each floor-*i* node *n* of type T such that $M(T) = ()$, if every type T node is unique with respect to $typeL(T)$, add $(T, typeL(T))$ to *M* and for each type T' that is a prefix of T, add (T', **suffix**$(T', T)/typeL(T)$) to *M*;
3) $i = i + 1$; terminate Phase 1 if the next floor is the root, or go to 2).

Phase 2: expand with non-local identifiers

Starting from the root and working down the tree, for each node *n* of type T such that *n* is not unique with respect to $typeL(T)$, add (T, *Id*) to *M* where *Id* is a list obtained by appending to $typeL(T)$ the identifier of *n*'s parent.

Fig. 4. Algorithm for computing identifiers

5 Semantic Change Detection

We are now able to semantically identify a node in an XML document. In this section, we discuss how nodes in different versions can be matched based on their identifiers. Once all semantically identical nodes are matched, we regard the unmatched elements as change participants.

5.1 Semantic Node Matching

The following definitions assume that element nodes *p* and *q* are both of type T and reside in different versions V_p and V_q of an XML document. ID(*p*) is *p*'s identifier.

Definition [Type Territory]. The territory of a type *T*, denoted T_T, is the set of all text nodes that are descendants of the least common ancestor, denoted $lca(T)$, of all of the type T nodes. ∎

Within the type territory is the territory controlled by individual nodes of that type.

Definition [Node Territory]. The territory of a type T node *p*, denoted N_p, is T_T excluding all text nodes that are descendants of other type T nodes. ∎

The type territory of T contains all the information that might be useful in identifying any type T node. The territory of a specific node of that type is contained in the type territory, but does not contain any node that is a descendant of another type T node.

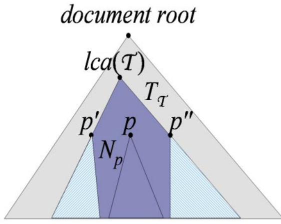

Fig. 5. Node territory

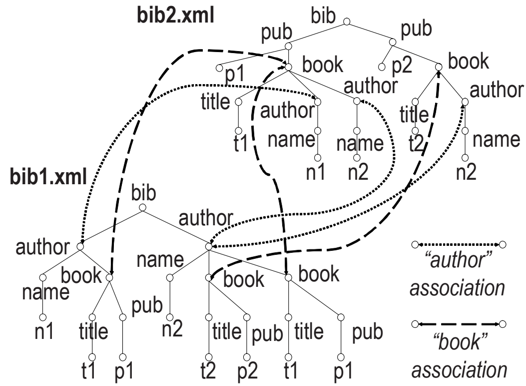

Fig. 6. Part of the match between bib1.xml and bib2.xml

Fig. 5 visualizes the idea of type territory and node territory. Suppose there are three type *T* nodes, *p*, *p*' and *p*''. The type territory of *T*, T_T, is the subtree rooted at the node *lca*(*T*). The node territory of *p*, N_p, is the area shaded dark in the figure. N_p is T_T excluding the two subtrees rooted at *p*' and *p*'' (represented by the striped areas).

The type territory of *book* in Fig. 5, for example, is (Dan Brown, The Da Vinci Code, Doubleday, $24.95, Angels & Demons, Pocket Star, $7.99). The node territory of the leftmost *book* node is (Dan Brown, The Da Vinci Code, Doubleday, $24.95).

Now we are ready to match nodes in successive versions.

Definition [Admits]. *q admits p* if $Eval(q, \mathrm{ID}(q)) \subseteq N_p$. ∎

In general, $Eval(p, \mathrm{ID}(p))$ is a list of lists because each XPath expression in the identifier evaluates to a list of values. Here in semantic matching we implicitly convert (flatten) it to a list of distinct values. This is because only the values are important in our semantic matching.

Definition [Node match]. Nodes *p* and *q* are *matched* if and only if *p* and *q* admit each other. ∎

Intuitively, admission and match can be described as follows. *q* is identified by a list of text values $q_1, \ldots, q_n$ in V_q. If a node *p* in V_p has at least as much information as *q* does, then *p* should have a group of text values $q_1, \ldots, q_n$ in its own territory N_p. A match implies semantic equality between two nodes, thus it requires admissions in both directions.

5.2 Semantic Matching for Sample Documents

We now show how nodes are matched based on the criteria described above. Fig. 6 shows bib1.xml and the next version, bib2.xml, in which there has been significant structural change. The document versions are displayed as tree diagrams rather than textually to better illustrate how the nodes are matched. The change between the two versions is similar to that presented in Fig. 1 and Fig. 3. Intuitively, we can tell that

the semantics of the two versions are the same insofar as they contain the same information, but arranged to different schemas.

First we need to compute the identifiers for each node in both versions. We are then able to evaluate the value of each node's identifier. For example, the values of <book>'s identifiers in both versions are shown in Table 1. The territory of the leftmost <book> in bib2.xml is (p1, t1, n1, n2, p2); the identifier of the leftmost <book> in bib1.xml evaluates to (n1, t1). Hence the leftmost <book> in bib1.xml admits the leftmost <book> in bib2.xml. Similarly, the first <book> in bib2.xml admits the first <book> in bib1.xml. Therefore, the first <book> in bib1.xml and the first <book> in bib2.xml match. This is represented in Fig. 6 by a dashed line connecting the two.

All matches for *book* and *author* are shown in Fig. 6. (Two matched *author* nodes are connected by a dotted line.) To preserve the clarity of the figure, we do not show matches for all of the nodes. It turns out that each node is matched to one or more nodes in the other version. For a change (insertion, deletion or update) to occur, there has to be at least one unmatched node in one version. We can thus conclude that there is no semantic change from bib1.xml to bib2.xml for *book* and *author* elements.

Table 1. Values of identifiers for *book*

bib1.xml	**Value of Identifier**
leftmost *book*	((n1), (t1))
middle *book*	((n2), (t2))
rightmost *book*	((n2), (t1))
bib2.xml	**Value of Identifier**
leftmost *book*	((t1))
rightmost *book*	((t2))

5.3 Complexity Analysis

Let P and Q denote numbers of nodes in two documents to which nodes p and q belong, respectively. Deciding whether node q admits node p is bounded by $O(P*\log(P))$. Thus deciding whether q and p match takes $O(P*\log(P)+ Q*\log(Q))$.

For entire document matching, it would take $O(P*Q*(P*\log(P)+ Q*\log(Q)))$ if we try to directly match all possible pairings of nodes in two documents. To reduce the time cost of matching, we use the following heuristic. If the identifier for a node type remains the same, then we can base our matching solely on the values of the evaluated identifiers, effectively skipping the expensive test to search for a match in the node's territory. The notion of node territory is crafted to help match nodes when their identifiers are different; two nodes match if each node's territory includes the other's evaluated identifier. However, if the identifier for a node type remains unchanged, we can in fact base our matching solely on the evaluated identifiers. In this case, two nodes match if and only if their evaluated identifiers are the same. There is no need to compute node territories. In most real-world applications, the extent of change over versions is usually small. Hence, it is reasonable to expect that the identifiers of most of the types remain the same over versions. Based on this assumption,

the number of direct matching attempts is often relatively small. For large documents, computing node territories will take a major portion of the processing time.

6 Conclusions and Future Work

This paper proposes a new approach to detect changes in successive versions of XML documents. The approach is novel because it utilizes semantic change; previous work focused on structural change. We first define the notion of a semantic identifier, which is an expression that serves to distinguish elements of a particular type. This paper then sketches an algorithm to efficiently compute these identifiers. Changes in successive versions are obtained by matching nodes based on their semantic identifiers. Our approach is to observe that the information that identifies an element is conserved across changes to a document. We provide an algorithm that matches a pair of nodes by looking for the identifying information of one node in the territory of the second node. The advantage of our approach is that we can match nodes even when there has been significant structural change to a document. Compared to conventional structural change detection, our semantics-based technique is able to detect semantic change in various conditions, without prior knowledge of schema.

The next stage of this research is to implement the algorithm in C and integrate the code into the Apache web server to support server-side versioning of XML documents [9]. Another task is to devise a metric to measure the quality of a semantic match, and to develop an algorithm to compute the metric mechanically. With this refined framework of semantics-based node association in place, we then plan to build a transaction-time XML repository and implement a system such as *TT*XPath [10] to support transaction-time XPath queries.

References

1. A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*. Oxford University Press, 1997.
2. B. Brewington and G. Cybenko. “How Dynamic is the Web?” In *Proc. of the 9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000, 257–276.
3. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. “Keys for XML”. In *Proc. of the 10th International World Wide Web Conference*, Hong Kong, China, 2001, 201–210.
4. G. Cobéna, S. Abiteboul, A. Marian. “Detecting Changes in XML Documents”. In *Proceedings of ICDE*, San Jose, February 2002, 41–52.
5. S. Chawathe and H. Garcia-Molina. “Meaningful Change Detection in Structured Data”. In *Proceedings of SIGMOD Conference*, June 1997, 26–37.
6. Cho, J. and H. Garcia-Molina. “The Evolution of the Web and Implications for an Incremental Crawler”. In *Proc. of VLDB Conference*, Cairo, Egypt, Sep. 2000, 200–209.
7. S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom. “Change Detection in Hierarchically Structured Information”. In *SIGMOD Conference*, Montreal, Canada, June 1996, 493–504.

8. F. Douglis, T. Ball, Y. F. Chen, E. Koutsofios. "The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web". *World Wide Web*, 1(1): 27–44, Jan. 1998.
9. C. Dyreson, H. Ling, Y. Wang. "Managing Versions of Web Documents in a Transaction-time Web Server". In *Proc. of the 13th International World Wide Web Conference*, New York City, May 2004., 421–432.
10. C. Dyreson. "Observing Transaction-time Semantics with TTXPath". In *Proceedings of WISE*, Kyoto, Japan, December 2001, 193–202.
11. Fabio Grandi. "Introducing an Annotated Bibliography on Temporal and Evolution Aspects in the World Wide Web". *SIGMOD Record*, Volume 33, Number 2, June 2004.
12. D. Gao and R. T. Snodgrass. "Temporal Slicing in the Evaluation of XML Queries". In *Proceedings of VLDB*, 2003, 632–643.
13. C. M. Hoffmann, M. O'Donnell. "Pattern Matching in Trees". *JACM*, 29: 68–95, 1982.
14. V. I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". *Cybernetics and Control Theory*, 10: 707--710, 1966.
15. L. Liu, C. Pu, R. Barga, and T. Zhou. "Differential Evaluation of Continual Queries". *Proc. of the International Conference on Distributed Computing Systems*, 1996, 458–465.
16. L. Liu, C. Pu, and W. Tang. "Continual Queries for Internet Scale Event-Driven Information Delivery". *IEEE Trans. Knowledge Data Engineering*, 11(4), 610–628, 1999.
17. S. Lu. "A tree-to-tree distance and its application to cluster analysis". *IEEE Trans. Pattern Analysis and Machine Intelligence*, 1(2): 219–224, 1979.
18. W. Masek, M. Paterson. "A faster algorithm for computing string edit distances". *J. Comput. System Sci.*, 1980, 18–31.
19. E. Myers. "An O(ND) Difference Algorithm and Its Variations". *Algorithmica*, 1(2): 251–266, 1986.
20. K. C. Tai. "The Tree-to-Tree Correction Problem". *JACM*, 26: 485–495, 1979.
21. "XML Path Language (XPath) 2.0". *W3C*, `www.w3c.org/TR/xpath20/`, current as of August 2004.
22. Y. Wang, D. DeWitt, J.-Y. Cai. "X-Diff: An Effective Change Detection Algorithm for XML Documents". `www.cs.wisc.edu/niagara/papers/xdiff.pdf`, current as of August 2004.
23. R. A. Wagner, M. J. Fischer. "The string-to-string correction problem". *JACM*, 21: 168–173, 1974.
24. K. Zhang and D. Shasha. "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems". *SIAM Journal of Computing*, 18(6): 1245–1262, 1989.
25. K. Zhang, R. Statman, D. Shasha. "On the Editing Distance between Unordered Labeled Trees". *Information Processing Letters*, 42: 133–139, 1992.
26. K. Zhang. "A Constrained Edit Distance between Unordered Labeled Trees". *Algorithmica*, 205–222, 1996.