Proceedings of the Second International Conference on Web Information Systems Engineering (WISE2001), Kyoto, Japan, December 2001, pp. 193-202.

Observing Transaction-time Semantics with *tt*XPath

Curtis E. Dyreson
School of Electrical Engineering and Computer Science
Washington State University, USA
cdyreson@eecs.wsu.edu

Abstract

Transaction time is the time of database transactions that create, modify, or destroy facts. It is used to record when facts exist in a database. Accounting for transaction time is essential to supporting audit queries that delve into past database states and differential queries that pinpoint differences between two states. In a web context, transaction time is a problematic concept because there are no transactions. Browsers and other consumers of web data can observe snapshots of resources like XML documents but are rarely active participants in their creation or destruction.

This paper presents the ttXPath data model and query language. ttXPath extends XPath with support for transaction time. XPath is a specification language for locations in an XML document. It serves as the basis for XML query languages like XSLT and XQuery. XPath has no temporal semantics. To construct a ttXPath data model, snapshots of an XML document are obtained over time by an observer. The snapshots are then merged and transaction times are associated with each edge and node. The ttXPath query language extends XPath with a transaction-time axis to enable a query to access past or future states, and with constructs to extract and compare times. ttXPath maximally reuses XPath hence the changes needed to support transaction time are minimal and ttXPath is fully backwards-compatible with XPath.

1. Introduction

The World-Wide Web ("web") is the largest, most frequently used, text-based information resource. The web currently has several million servers providing access to several billion documents. Many of these documents conform to the HyperText Markup Language (HTML), but in the near future, the Extensible Markup Language (XML) [32] is expected to replace HTML as the mark-up language of choice for web documents [3, 11]. XML is also expected to become an important language for web data exchange.

The database research community has been active in applying database concepts and techniques to the web [14]. The expected growth of data encoded in XML has inspired several new data models and query languages. Semistructured and unstructured data models, based on graph representations of data, have been advocated for managing and querying data that lacks a rigid schema, such as data encoded in XML [1, 6, 7, 14, 20, 24, 28]. Previous research has shown that much of XML can be mapped to and from a semistructured data model [28].

Over the past two decades there has been a substantial amount of research on extending databases to support time [17, 21, 27, 26, 29]. This research has led to the development of *transaction-time databases* [19, 22, 25]. Transaction time is the time when a particular fact is stored in a database and considered current, i.e., the time between when it is inserted and deleted (an update is modeled as a deletion followed by an insertion). Very briefly, a transaction-time database stores all of the past states of a database and allows queries, called transaction timeslice, to retrieve any desired past state.

This paper presents an XML query language and data model that supports transaction time called *tt*XPath. There are many proposed query languages for XML [1, 4, 20, 28, 30, 31, 33, 34]. All of these proposals lack temporal semantics. We have previously described and implemented an SQL-like query language called AUCQL for a semistructured database that includes support for transaction time [13]. In this paper we extend XPath [30] with concepts borrowed from AUCQL. XPath is a specification language for locations in an XML document. It serves as the basis for XML query languages like XSLT [31] and XQuery [34].

Transaction time is a problematic concept for the web because there are no transactions. Browsers and other consumers of web data have read access to data, but rarely can insert, update, or delete data. Updates to web data are irregular, ad-hoc, and hidden from readers of that data. To remain current with a constantly evolving data source, the

data is occasionally re-read and replaces or adds to a local data store. We call such a reader an *observant system*. An observant system is a system that can observe data but (generally) cannot modify it. A web browser is an observant system. It reads data from the web but cannot update that data (HTTP PUTs are rare). A web server (an HTTP server) is also an observant system [12]. A web server responds to an HTTP GET by reading a resource from local storage, but it is uninvolved in an update of that resource. Observant systems are easy to deploy on the web but must detect updates during a read by comparing the current state with the last observed state of the same resource. A difference denotes that an update occurred sometime in the past.

In Section 2 an example is given to motivate the utility of this research. Next the time and data model for *TT*XPath are presented. The *TT*XPath data model is a snapshot data model that extends the XPath data model. In Section 5, the temporal query language extensions are sketched and several example queries are described. The paper then presents related work and concludes.

2. Motivating Example

To exemplify our data model, consider the following scenario. `registrar.wsu.edu` is a site that offers XML data on courses taught during the current semester at Washington State University (WSU). New courses are added each semester to the site and old courses are removed. Updates to repair incorrect information like misspellings are infrequent. `studentlife.wsu.edu` is a portal that provides information for students at WSU. Some of the information comes from observing the data available at `registrar.wsu.edu`. Several times each semester, course data is uploaded from `registrar.wsu.edu` to `studentlife.wsu.edu`. In Fall, 2001, the following XML was uploaded (the XML has been kept very simple for expository purposes).

```
<?xml version="1.0">
<registrar>
  <department>
  <course>
    <prerequisite>CS223</prerequisite>
    <code>CS451</code>
    <teacher>Julie</title>
  </course>
  </department>
</registrar>
```

Note that the above data does not have any *explicit* temporal information, that is, there are no timestamps in attribute values or in element content.

Figure 1 sketches the (logical) tree structure of the XML fragment (excluding whitespace). The current course information is displayed in the tree in the foreground. Each node in the tree corresponds to an element or value in the document. The tree in the background is the old course information for Spring, 1999. For the current semester, information about a prerequisite course has been added and the teacher updated.

Suresh is a student at WSU. He would like to take a computer science course as an elective to complete his degree. Unfortunately Suresh has taken all of the computer science courses offered in the previous three years. If there are no new computer science courses he will have to take a Math course instead. To find out about new courses, Suresh visits the student life portal and would like to pose the following query (in the appropriate XML query language): "Are there any courses that are offered this year that have not been offered in the past two years?" To evaluate the query, current information will have to be compared against that available in the past. This kind of query is called a *differential* query. Differential queries are useful in many contexts. Suresh may be concerned about his finances and wish to identify trends in stocks that he owns. Or he may be a Napster user and would like to find the songs added since his previous visit to the Napster web site.

Suresh is also interested in retrieving changes made to his grades for the Spring, 1999 semester. `register.wsu.edu` currently shows that he earned a B in programming, but Suresh is pretty sure that he got an A. He wonders if and when the grade was changed. Suresh would like to make an *audit* query. An audit query rolls the data back to some past state, in Suresh's case, to Summer, 1999 when Spring grades were posted. Ideally, the query will also generate an *audit trail* that shows changes over time to selected data.

The key contribution of this paper is that it describes how to support the kinds of queries of interest to Suresh on data that lacks explicit timestamps in an observant system. We could assume that explicit timestamps were present in the data (this is the starting assumption in temporal database research). Alternatively we could suggest several rigorous strategies for specifying such timestamps [15]. But in this paper we do not make such an assumption nor do we discuss alternatives because our focus is on the current state of XML. Lots of XML data currently exists without any explicit timestamps. Even when a timestamp is present, the structure and semantics of the timestamp will likely vary from document to document since none of the XML schema languages currently proposed have temporal semantics [18].

Although there are no explicit timestamps in most XML data, an observant system has access to at least two *implicit* timestamps. If the XML is stored in a file (which is often the case), then the file's modification time is a timestamp that denotes the earliest time at which that data existed. The HTTP 1.1 protocol provides a mechanism for recovering a

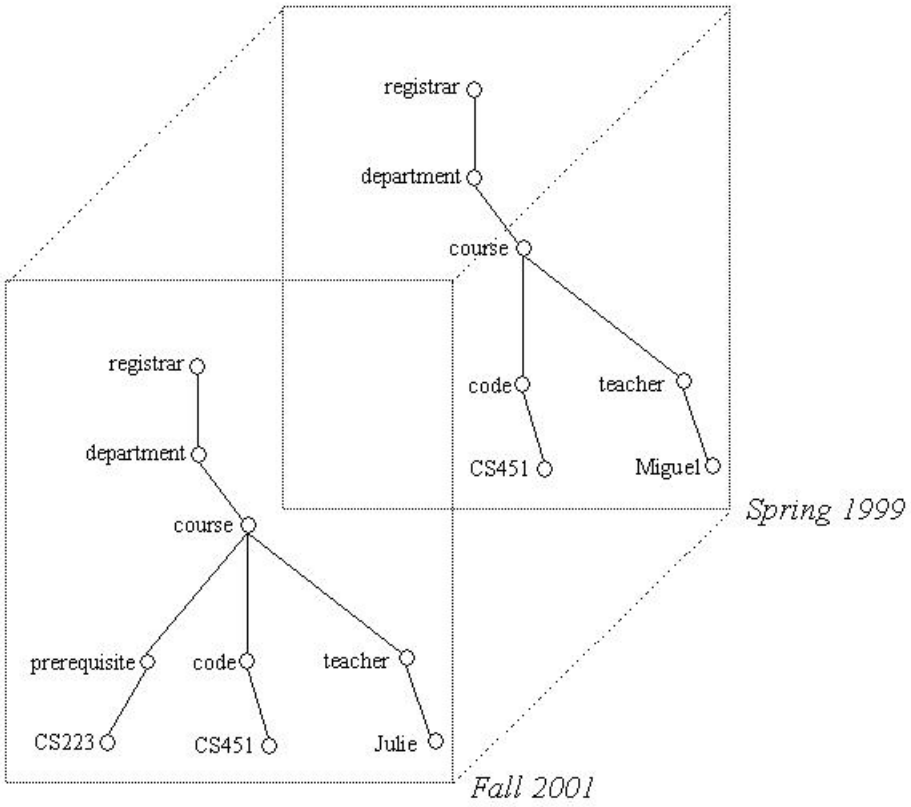

Figure 1. Two snapshots of the XPath data model for the example fragment

requested resource's modification time (which is important for caching). The second implicit timestamp is the time at which the data is read by the observer.

Both the read and modification timestamps are from the *transaction time* domain. That is, they concern the time(s) when the data existed at the data source and when it was retrieved by a browser. In contrast a *valid time* timestamp would denote when data is true in some modeled reality [16]. In the above scenario involving course data, the distinction between these two different kinds of time evaporates since it is assumed that data is available from the server only when that data is valid. Said differently, a course is added to the server when it is currently being offered, and immediately removed once it is no longer offered.

The technique proposed in this paper is to utilize the implicit timestamps to build a transaction-time history for a document. Figure 2 shows the transaction-time history for the example data in Figure 1. In Spring, 1999 the document is created. In Summer, 2000 the observant system read an unchanged version of the document (the file modification time was still Spring, 1999). But in Fall, 2001, it was read again and the file modification time was Fall, 2001 (the document was observed as soon as it was modified). Since the document was accessed only three times, most of the history is unobserved, and therefore has to be inferred. The area shaded gray depicts the *known* history. Information is known if it is unchanged between observations. The areas **not** shaded gray represent an *assumed* history. In this paper we make the following assumption, which we call the *continuity assumption*: information is assumed to exist until explicitly modified. The teacher data about Miguel was known to exist from Spring, 1999 to Summer, 2000. Since the document was not read between Summer, 2000 and Fall, 2001 we assume that Miguel remains the teacher until the document was modified in Fall, 2001 to insert Julie as the teacher. Because of the continuity assumption, the status of Miguel being a teacher is recorded as *assumed*, rather than known from Summer, 2000 to Fall, 2001. In Figure 2, the assumed data is the area stippled with dots. The continuity assumption can be further classified as *optimistic* or *pessimistic*. The optimistic continuity assumption is that every modification of the document is observed. The pessimistic continuity assumption is that the observant system might miss some modifications. In the context of the exam-

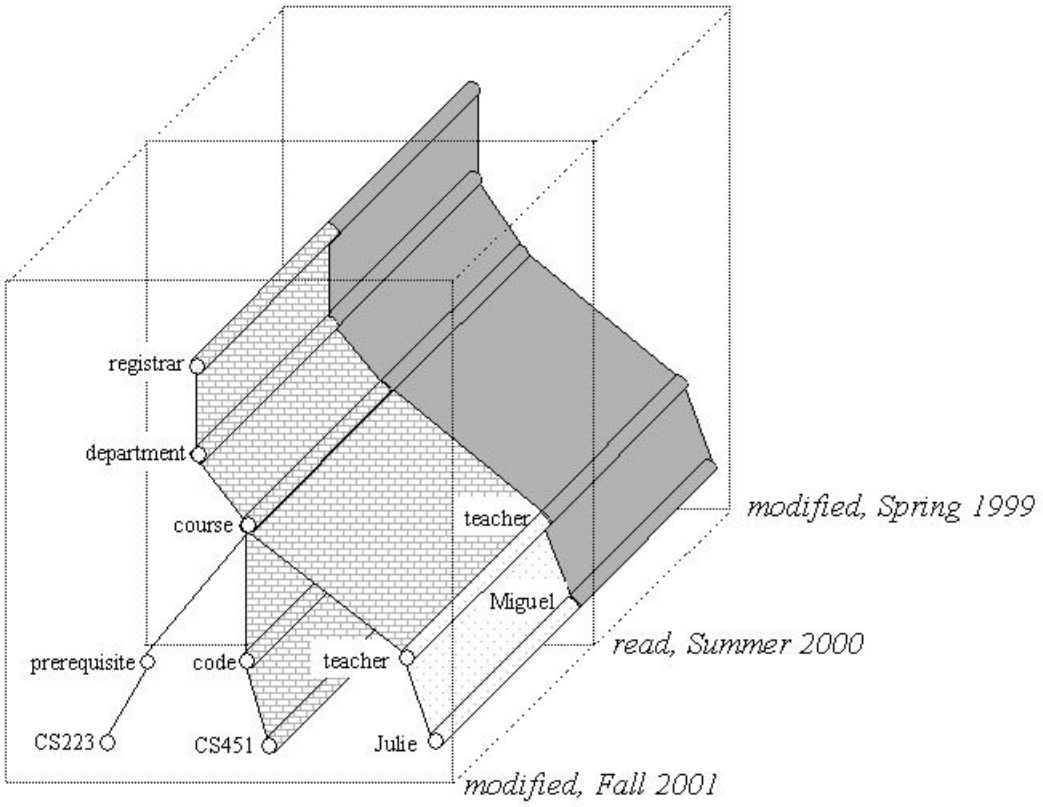

Figure 2. The *TT*XPath data model for the example fragment

ple, if the observant system adopts an optimistic continuity assumption then the data was modified only in Spring, 1999 and Fall, 2001. This implies that the portions of the tree that remain unchanged in both modifications should be known rather than assumed. But with a pessimistic continuity assumption the data could have been arbitrarily modified many times without being observed. In Figure 2 the area with a brick pattern has a known status under an optimistic continuity assumption, but is assumed under a pessimistic continuity assumption, since this portion of the tree did not change in the second modification. The area with a dot pattern however is assumed data under both the optimistic and pessimistic continuity assumptions since it was certainly changed by some modification. In the remainder of this paper, we will adopt the pessimistic continuity assumption. Section 4 describes the data model in detail. Finally, the lifetime of each node in Figure 2 is depicted as a pipe. In a *TT*XPath query, a pipe can be traversed to access past or future versions of a node. Section 5 discusses the query language and gives many example queries. In the next section, the time model we use in the paper is introduced.

3. Time Model

An observant system only occasionally observes a document. Each observation yields information about the document as it exists at a single point in time, which we will call the *read time*. The observation may also yield meta-data about the document. In the HTTP 1.1 protocol an important piece of meta-data that is obtained is the *modification time* of a document. The modification time is the time when the document was last modified. We will call each observed modification a new *observed version* of the document.

Notationally, we will represent the history of an XML document, X, as a sequence, $X_{m_1}^{r_1}, \ldots, X_{m_n}^{r_n}$ where m_i is the modification time of observed version i and r_i is the (most recent) read time. Subsequent reads on an unmodified version will push the read time later. Without loss of generality, we restrict the discussion to a single XML document from a single server since a document history with nested sub-documents resident on different servers can be viewed as a single document with a global history of reads and modifications on any of the sub-documents.

The read and modification times are used to construct

a useful clock called the *version clock*. The version clock keeps time in the observant system. The modification time comes from the clock on the file system at which the resource resides, while the read time is from the observant system's clock. (It does not matter if these clocks are synchronized, however, we stipulate that if the read time of an observed version is earlier than the modification time, then it be adjusted to the modification time.) The version clock adds one clock tick for each version, and tells time by reporting the modification and read times for that version.

Definition 3.1 The version clock, $C_{X_h}(t)$, is a partial function constructed from the history of a well-formed XML document, $X_h = X^{r_1}_{m_1}, \ldots, X^{r_n}_{m_n}$, as follows: $C_{X_h}(t) = (m_t, r_t)$. □

The read time and modification time can be used to infer knowledge about unobserved states of the document. If the document has not been modified since the last read, then it is *known* that the current observed version is the version in existence since the modification time. If the document has been modified since the last read, then the evolution of the document is unknown between the read time of the previous observed version and the modification time of the current observed version. One or more transitory versions may have existed during that time which the observant system missed. An optimistic observer will *assume* that no versions were missed. We distinguish between known and assumed versions of a document.

The read and modification times are kinds of transaction time. Research in temporal databases has identified two primary, distinct time dimensions: *valid time* and *transaction time* [16]. Valid time is the real-world time of a fact, whereas transaction time is the database time when that fact was present in the database.

In this paper, the transaction-time domain is a set of instants, $T_{TT} = \{0, \ldots, uc, \ldots, \infty\}$. The *until changed* variable, *uc*, represents the ever-changing current time [9]. In contrast to traditional temporal database research, the transaction-time domain ends at ∞ thus permitting *future* transaction times. This enables document authors to set expiration times for documents and to schedule documents for future release.

In this paper, example times will be represented using Gregorian calendar conventions in the granularity of days, so each instant in the transaction-time domain corresponds to a day. In practice, the literal representation and granularity of times in the transaction-time domain is system dependent, with a granularity of UTC seconds and a Gregorian calendar representation being common.

4. Data Model

A well-formed XML document is a collection of nested *elements*. An element begins with a start tag and ends with a paired end tag. Between the tags, an element might contain *content*, that is, text or other elements. The XPath data model is commonly assumed to be an ordered tree. The tree represents the nesting of elements within the document, with elements corresponding to nodes, and element content comprising the children for each node. Unlike a tree, the children for a node are *ordered* based on their physical position within the document.

Unfortunately the XPath recommendation [30] does not provide a formal model. Below we give one possible model that omits details extraneous to the aims of this paper.

Definition 4.1 The XPath data model, D, for a well-formed XML document, X, is a four-tuple, $D[X] = (r, V, E, I)$, where the tuple members are defined below.

- V is a set of nodes of the form (i, v) where v is the node identifier and i is an ordinal number such that for all $(i, v), (j, w) \in V$, v starts before w in the text of X iff $i < j$.
- E is a set of edges of the form (n, v, w), where $v, w \in V$ and n is an ordinal number that captures the ordering among the edges emanating from v. An edge (i, v, w) means that v is a parent of w. In terms of X, it represents that w is in the immediate content of v. Among the children of v, y is before z in X iff for $(i, v, y) \in E$ and $(j, v, z) \in E$, $i < j$.
- The graph, (V, E), forms a tree.
- I is the information set function which maps a node identifier to an *information set*. An information set is a collection of properties that are generated during parsing of the document. For example, an element node has the following properties: *Value* (the element's name), *Type* (element), and *Attributes* (a set of name-value pairs, in XPath, attributes are unordered). In future, it may be possible to dynamically extend the information set (the XML Information Set proposal is available from the W3C but is undergoing extensive development).
- $r \in V$ is the root, i.e., $\neg\exists(i, x, r) \in E$. r is the data model root, rather than the document root (the first element in the document) since a document may contain processing instructions and comments before the first element. □

The TTXPath data model is an extension of the XPath data model. To capture transaction time, we define a simple snapshot data model.

Definition 4.2 The *TT*XPath data model, D_{TT}, is constructed from the history of a well-formed XML document, $X_h = X^{r_1}_{m_1}, \ldots, X^{r_n}_{m_n}$, where m_i is the modified time of version i and r is the read time. The data model is $D_{TT}[X_h] = H(t)$, where $H(t)$ is a partial function that maps a transaction time, t, to an ordered pair (s, D).

- D is the XPath data model that was in existence (either known or assumed) at time t. D is referred to as the *snapshot* of X_h at time t.
- s is the status which is either *known* or *assumed*. The status is *known* if there exists some $X^{r_i}_{m_i}$ such that $m_i \leq t \leq r_i$. The status is *assumed* for all other values of t between m_1 and $uc + 1$. $H(t)$ is undefined for all other values of t. □

There are two things to observe about this data model. First, it is a linear, not a *branching*, history of versions, A branching history is one where more than one version of a document is considered current simultaneously, perhaps due to the independent editing of local copies by multiple editors. Second, each snapshot is an XPath data model. The XPath query language, and query languages that use XPath, are well-defined for snapshots.

The drawback of the snapshot data model is that, if naively implemented, it would need a lot of space. A document may change very little when it is modified. One way to recoup some space is to merge versions by identifying those portions of the data model that are unchanged by a modification. Identifying which portions remain the same is a challenging problem. Nguyen et al. have developed an efficient XML 'diff' technique to (approximately) identify changes [10]. Another approach is to track changes by defining (persistent) keys for elements [5]. We assume that some method exists for identifying changes in nodes and edges between versions and utilize that method to build a space-reduced data model. The space-reduced model simply merges information common to consecutive snapshots, and records the lifetime of each node and edge with a transaction-time interval.

5. Query Language

In this section we introduce the *TT*XPath query language. *TT*XPath extends XPath with constructs to query the transaction time. The language is designed to meet several goals. First and foremost, it has to be fully compatible with XPath. Existing and future XPath queries should work exactly the same in *TT*XPath. A secondary goal is that the extensions should be both minimal, to reduce the conceptual overhead, and efficient. Our design reuses XPath whenever possible. The general idea is that transaction time will be used to choose a snapshot, and XPath used to query that snapshot. A third goal is that the extensions should include all "useful" constructs. Those constructs should be favored in abbreviated syntax. Of lesser importance, at this time, are the completeness and expressiveness of the query language. In future, we will address these properties.

The presentation in this section is somewhat backwards. First, examples of *TT*XPath are given. Next a brief overview of XPath is given followed by a detailed discussion of the semantics of *TT*XPath. The reason that we give the examples first is to familiarize the reader with the ease of giving *TT*XPath queries.

5.1. Examples of *TT*XPath

Let's look at some example queries. Recall that *TT*XPath is used to specify locations in an XML document. List course nodes as of Spring, 1999 (note that all *TT*XPath queries occupy a single line of text, but due to column width restrictions in this paper *TT*XPath queries are listed on multiple lines, please ignore the additional whitespace).

```
/tt-past::slice('Spring, 1999')
  /department
    /course
```

In the above query, only nodes described as a `department/course` in the version as of Spring, 1999 are retrieved. In the following query, we use the current schema to query for course nodes that existed in the database since Spring, 1999 (but may have been described differently, e.g., as `school/subject`).

```
/department
  /course
    /tt-past::slice('Spring, 1999')
```

The difference between the two queries is subtle. The first query rolls the document back to the version as of Spring, 1999 and then executes an XPath query within that version. The second query finds course nodes that are in the current version and then rolls back those nodes to their Spring, 1999 counterparts (if they exist). A course that was deleted in Fall, 2000 would appear in the result of the first query, but not in that of the second query; while a course that was reachable via `school/course` in the Spring, 1999 version of the document is (possibly) in the result of the second query, but not the first.

List courses offered in Spring, 1999, but not in the following semester (see next page).

```
/tt-past::'Spring, 1999'
  /department
    /course
      [not(tt-future::'Summer, 1999')]
```

This query rolls the database back to Spring, 1999. Then for each course found it tests to determine if that course can be rolled forward to Summer, 1999. If the course has been deleted then there is no version of it in the Summer, 1999 snapshot. Note that the `not` operator in XPath evaluates to true if the node-set is empty.

List courses that are currently offered that were also offered in Spring, 1999.

```
/department
  /course
    [tt-past::'Spring, 1999']
```

This query finds current courses and then tests to determine if some version of the course was offered in Spring, 1999.

List courses that have not been revised since Spring, 1999.

```
/tt-past::'Spring, 1999'
  /department
    /course
      [not(tt-future::tt-next())]
```

This query rolls the document back to the Spring, 1999 snapshot. It then finds course information. For each course it tests to see if any new version of the course exists; the `tt-future` axis just switches the temporal direction.

List courses currently offered that have been added since Spring, 1999.

```
/department
  /course
    [tt-earliest() &gt; 'Spring, 1999']
```

The query first finds current courses. For each node found it then extracts the time of the earliest version and ensures that it is greater than Spring, 1999.

List courses that have been dropped since Spring, 1999.

```
/tt-past::'Spring, 1999'
  /department
    /course
      [not(tt-future::slice('now')]
```

This query first finds course information as of Spring, 1999. The course has been dropped if it is not possible to roll the past course node forward to now because that indicates the course has been deleted sometime prior to now.

In general, an audit trail is not possible to construct in *TT*XPath directly since the trail could include an unbounded number of revisions. But in an XSLT template, *TT*XPath can be used to generate all revisions. The following XSLT template uses the `tt-next()` node test to iterate over all previous versions.

```
<xsl:template
  match="department/course">
   <xsl:copy-of select="."/>
   <xsl:apply-templates
     match="tt-past::tt-next()"/>
</xsl:template>
```

5.2. XPath

Before describing *TT*XPath, we briefly summarize XPath. An XPath query is a sequence of steps. Each step consists of four parts: a *context*, an *axis*, a *node test*, and a list of *predicates*. The *context* is the environment, including the context node, in which the step begins evaluation. The *axis* specifies a set of nodes, relative to the context node, that might be in the result of the step. Possible axes include `self`, `parent`, `child`, `descendant`, `ancestor`, `descendant-or-self`, etc. The *node test* is a predicate that is applied to each node in the axis. Possible node tests include `any` and `element()`. The node test is syntactically separated from the axis with the string '`::`'. Those nodes that pass the node test are then tested by the predicate(s). A step may have several predicates, each denoted by brackets. To qualify for a result, a node must pass every predicate. A predicate may itself include one or more XPath queries. A simple syntax for a step is given below.

$$axis\texttt{::}node\ test\texttt{[}predicate_1\texttt{]}\ldots\texttt{[}predicate_n\texttt{]}$$

The result of a step is an *ordered* list of nodes, called, paradoxically, a *node-set*. The ordering is based on the order in which the nodes appear in the document. The direction, *document order* or *reverse document order* relative to the context node, is determined by the axis (e.g., `child` is document order while `ancestor` is reverse). The result of a query is the result of the final step in the query. Nodes in the result of non-final steps are used (in order) as the context node for the next step. Syntactically, the steps are separated by the '`/`' character. A simple syntax for a query is given below.

$$/step_1/\ldots/step_m$$

An example query to retrieve the children of the `course` element with a `code` attribute value of `CS451` is given below.

```
/descendant-or-self::course
  [attribute::code="CS451"]/child::*
```

The first step explores the `descendant-or-self` axis from the data model root. It applies an element test to keep only `course` elements. The predicate filters those nodes that lack a `code` attribute of `CS451`. The second step follows the `child` axis and retrieves any node (the wildcard is `*`).

XPath has an abbreviated syntax that shortens most queries. A shorter, semantically-equivalent query using the abbreviated syntax is given below.

```
//course[@code="CS451"]/*
```

Readers interested in further details should consult the XPath recommendation [30].

5.3. *TT*XPath

*TT*XPath extends each part of XPath. It adds several new axes, node tests, and temporal constructors. In addition, the query context is expanded to include the *current time*, the *reference time*, and the *temporal order*. The current time is the time at which the query is initiated. The reference time is the time, relative to a step, that is considered the time "now". Initially, the reference time and the current time are the same. But in a *TT*XPath query, the query's point of reference can be shifted to the past or the future by a temporal node test. The reference time is critically important because it specifies which snapshot to use in the evaluation of each XPath piece in a *TT*XPath query. The temporal order controls the ordering of nodes in a node-set (earliest or latest first, if it contains nodes from more than one version).

In the following sections we present each change in more detail.

5.3.1 Transaction-time axes

The following additional axes are available in *TT*XPath. Each axis specifies a *temporal direction* to search for versions (past or future), the desired *status* of the versions selected (known or both known and assumed), and a *temporal order* for the resulting node-set.

- `tt-past` — Select all past versions of the context node and set the temporal order to latest first.
- `tt-past-known` — Select only known past versions of the context node and set the temporal order to latest first.
- `tt-future` — Select only known, future versions of the context node and set the temporal order to earliest first.
- `tt-future-known` — Select all future versions of the context node and set the temporal order to earliest first.

Formally, the transaction-time axes have the meanings given in Figure 3. We assume that v is the context node, t_r is the reference time, t_c is the current time, and d is the temporal order. The context is inherited from the environment.

5.3.2 Transaction-time node tests

*TT*XPath adds several node tests, `slice()`, `tt-next()`, `tt-next-immediate()`, `tt-previous()`, and `tt-previous-immediate()`, that are supported only along a transaction-time axis.

The `slice()` test is temporal timeslice. It takes as input a transaction-time point literal and selects the version of the node current at the time specified.

$$\texttt{slice}(t)[v, t_r, t_c, d] = \{(i, v)[v, t, t_c, d] \mid (t, v) \in \text{axis}\}$$

As a side effect `slice()` sets the reference time to the time of the slice (t replaces t_r in the result). In abbreviated syntax, if no node test is specified, `slice()` is used.

The remaining node tests are mostly useful for generating audit trails. The `tt-next()` node test chooses the version of the node that is current from among the descendents of the context node in the snapshot corresponding to the modification time of the "next" version on the version clock. The interpretation of "next" depends on the temporal order. If the direction is latest then "next" means earlier, if it is earliest then it means later. Let $R(t, v)$ be a partial function that extracts the subtree rooted at v at time t from D_{TT}, then `next-version()` is defined below. The context node, reference time, current time, and temporal order are inherited from the environment.

$$\begin{aligned}\texttt{next-version}(t)[v, t_r, t_c, \text{latest}] \\ = \{(i, v)[v, t, t_c, \text{latest}] \mid (t, v) \in \text{axis} \\ \wedge\ (t_r, v) \in \text{axis} \\ \wedge\ t < t_r \\ \wedge\ R(t) \neq R(t+1)\}\end{aligned}$$

As a side effect the reference time is set to the time of the next version.

The remaining node tests are similar. The `tt-previous` tests operate in the reverse temporal order, while the `immediate` variants only consider changes to the node and its children rather than all descendants.

5.3.3 Transaction-time constructors

Several temporal constructors are added. Their names and functionality are described in Table 1. Each constructor operates on the context node and extracts a transaction-time literal.

The constructors are used to extract the transaction time of a node when needed.

6. Related Work

There has been relatively little research in representing and querying time on the web. Chawathe et al. were the

$$\texttt{tt-past}[v, t_r, t_c, d]$$
$$= \{(t, v)[v, t, t_c, \text{latest}] \mid 0 \leq t \leq t_r \ \wedge \ H(t) = (\_, D) \ \wedge \ D = (\_, V, E, \_) \ \wedge \ v \in V\}$$
$$\texttt{tt-past-known}[v, t_r, t_c, d]$$
$$= \{(t, v)[v, t, t_c, \text{latest}] \mid 0 \leq t \leq t_r \ \wedge \ H(t) = (known, D) \ \wedge \ D = (\_, V, E, \_) \ \wedge \ v \in V\}$$
$$\texttt{tt-future}[v, t_r, t_c, d]$$
$$= \{(t, v)[v, t, t_c, \text{earliest}] \mid t_r \leq t \leq t_c \ \wedge \ H(t) = (\_, D) \ \wedge \ D = (\_, V, E, \_) \ \wedge \ v \in V\}$$
$$\texttt{tt-future-known}[v, t_r, t_c, d]$$
$$= \{(t, v)[v, t, t_c, \text{earliest}] \mid t_r \leq t \leq t_c \ \wedge \ H(t) = (known, D) \ \wedge \ D = (\_, V, E, \_) \ \wedge \ v \in V\}$$

Figure 3. Semantics of the transaction time axes

Constructor	Function
`tt-timeOf()`	Extract the reference time
`tt-first()`	Extract the modification time of the earliest version of a node
`tt-last()`	Extract the read time of the last known version of a node
`tt-modified()`	Extract the modification time of the current version of the node
`tt-read()`	Extract the latest read time of the current version of a node
`tt-last-modified()`	Extract the latest modification time for a node
`tt-last-read()`	Extract the latest read time for a node
`tt-previous-modified()`	Extract the modification time of the previous version of this node, sensitive to the temporal order
`tt-previous-read()`	Extract the read time of the previous version of this node, depends on the temporal order
`tt-next-modified()`	Extract the modified time of the next version of this node, sensitive to the temporal order
`tt-next-read()`	Extract the read time of the next version of this node, sensitive to the temporal order

Table 1. Temporal constructors in *TT*XPath

first to study time in an XML-like setting [8]. They encoded times in edge labels in a semistructured database and extended the Lorel query language with temporal constructs. Dyreson et al. extended their research with collapsing and coalescing operators [13]. The focus of this paper is on XPath and observant systems rather than semistructured databases. Grandi and Mandreoli present techniques for adding explicit valid-time timestamps in an XML document [15]. More recently, Amagasa et al. have presented a temporal extension of the XPath data model [2]. They studied valid time rather than transaction time in a non-observant system. They encoded the valid time by adding timestamps to edges rather than nodes and edges. Nguyen et al. have researched and developed an observant system: a data warehouse for XML data [23]. They focus primarily on techniques for keeping data fresh in the warehouse and on integrating updates when they arrive. Towards this end, they present an SQL-like query language for rule-based triggering and monitoring of updates.

7. Conclusion

Accounting for transaction time is essential to supporting audit queries that delve into past database states and differential queries that pinpoint differences between two states. Somewhat surprisingly, transaction time can be implemented in an observant system that has only read access to data. Updates can be detected *post facto* by tracking changes in observed states. The detection does not always yield certain information, and it is important to distinguish between states that are known and those that are assumed.

This paper sketches the *TT*XPath data model and query language. *TT*XPath extends XPath with support for transaction time. XPath has no temporal semantics. To construct a *TT*XPath data model, snapshots of an XML document are obtained over time by an observer. The snapshots are then merged and transaction times are associated with each edge and node. The *TT*XPath query language extends XPath with temporal axes, node tests, and constructors. The language extensions enable temporal queries to be naturally expressed. One important feature of *TT*XPath is that it reuses

XPath. *TT*XPath is fully backwards-compatible with XPath.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal of Digital Libraries*, 1(1):68–88, 1997.

[2] T. Amagasa, Y. Masatoshi, and S. Uemura. A Data Model for Temporal XML Documents. In *Database and Expert Systems Applications, 11th International Conference, DEXA 2000*, pages 334–344, London, UK, September 2000.

[3] T. Berners-Lee. Keynote Address. In *Seventh International World Wide Web Conference (WWW7)*, Brisbane, Australia, April 1998.

[4] A. Bonifati and S. Ceri. Comparative Analysis of Five XML Query Languages. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(1):68–79, 2000.

[5] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. C. Tan. Keys for XML. In *Proceedings of the Tenth International Conference on World Wide Web Conference (WWW10)*, pages 201–210, Hong Kong, China, May 2001.

[6] P. Buneman, S. Davidson, and D. Suciu. Programming Constructs for Unstructured Data. In *DBPL-5*, Gubbio, Italy, 1995.

[7] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Quebec, Canada, 4–6 June 1996.

[8] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *Proceedings of the International Conference on Data Engineering*, pages 4–13, Orlando, FL, February 1998. IEEE.

[9] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of *now* in Temporal Databases. *ACM Transactions on Database Systems*, 22(2):215–254, June 1997.

[10] G. Cobena and S. a. A. M. Abiteboul. Detecting Changes in XML Documents. Technical Report 194, Verso, France, March 2001.

[11] D. Connolly, R. Khare, and A. Rifkin. The Evolution of Web Documents: The Ascent of XML. *XML special issue of the World Wide Web Journal*, 2(4):119–128, Autumn 1997.

[12] C. Dyreson. Towards a Temporal World-Wide Web: a Transaction-Time Web Server. In *Proceedings of the Australian Database Conference (ADC '01)*, pages 290–301, Gold Coast, Australia, January 2001.

[13] C. Dyreson, M. Böhlen, and C. S. Jensen. Capturing and Querying Multiple Aspects of Semistructured Data. In *Proceedings of the International Conference on Very Large Databases (VLDB '98)*, pages 290–301, Edinburgh, Scotland, September 1999. http://www.eecs.wsu.edu/ cdyreson/AUCQL/.

[14] D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, September 1998.

[15] F. Grandi and F. Mandreoli. The Valid Web: it's Time to Go. Technical Report 46, TimeCenter, Aalborg, Denmark, December 1999.

[16] C. Jensen and C. D. (eds.). *A Consensus Glossary of Temporal Database Concepts - February 1998 Version*, pages 367–405. Springer-Verlag, 1998.

[17] N. Kline. An Update of the Temporal Database Bibliography. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(4):66–80, Dec. 1993.

[18] D. Lee and W. Chu. Comparative Analysis of Six XML Schema Languages. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(3):76–87, 2000.

[19] D. Lomet and B. Salzberg. *Transaction-Time Databases*, chapter 16, pages 388–417. Benjamin/Cummings, 1993.

[20] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Datat with FLORID: A Deductive Object-Oriented Perspective. *to appear in Information Systems*, 1998.

[21] E. McKenzie. Bibliography: Temporal Databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 15(4):40–52, December 1986.

[22] E. McKenzie and R. Snodgrass. Extending the Relational Algebra to Support Transaction Time. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 467–478, San Francisco, CA, May 1987. Association for Computing Machinery.

[23] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML Data on the Web. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, June 2001. Association for Computing Machinery.

[24] D. Quass, A. Rajaraman, J. D. Ullman, J. Widom, and Y. Sagiv. Querying Semistructured Heterogeneous Information. *Journal of Systems Integration*, 7(3/4):381–407, 1997.

[25] J. F. Roddick and R. T. Snodgrass. Transaction Time Support. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 17, pages 319–325. Kluwer Academic Publishers, 1995.

[26] M. D. Soo. Bibliography on Temporal Databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(1):14–23, Mar. 1991.

[27] R. Stam and R. T. Snodgrass. A Bibliography on Temporal Databases. *Database Engineering*, 7(4):231–239, Dec 1988.

[28] D. Suciu. Semistructured Data and XML. In *to appear in Proceedings of the International Conference on the Foundations of Data Organization (FODO '98)*, 1998.

[29] V. Tsotras and A. Kumar. Temporal Database Bibliography Update. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(1):41–63, March 1996.

[30] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, Nov. 1999.

[31] W3C. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, Nov. 1999.

[32] W3C. Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/REC-xml, Oct. 2000.

[33] W3C. The XML Query Algebra. http://www.w3.org/TR/query-algebra, Feb. 2001.

[34] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, Jun. 2001.