Temporal Slicing in the Evaluation of XML Queries

Dengfeng Gao   Richard T. Snodgrass

Department of Computer Science, University of Arizona, Tucson, AZ 85721
{dgao, rts}@cs.arizona.edu

Abstract

As with relational data, XML data changes over time with the creation, modification, and deletion of XML documents. Expressing queries on time-varying (relational or XML) data is more difficult than writing queries on nontemporal data. In this paper, we present a temporal XML query language, τXQuery, in which we add valid time support to XQuery by minimally extending the syntax and semantics of XQuery. We adopt a stratum approach which maps a τXQuery query to a conventional XQuery. The paper focuses on how to perform this mapping, in particular, on mapping *sequenced* queries, which are by far the most challenging. The critical issue of supporting sequenced queries (in any query language) is time-slicing the input data while retaining period timestamping. Timestamps are distributed throughout an XML document, rather than uniformly in tuples, complicating the temporal slicing while also providing opportunities for optimization. We propose four optimizations of our initial maximally-fragmented time-slicing approach: selected node slicing, copy-based per-expression slicing, in-place per-expression slicing, and idiomatic slicing, each of which reduces the number of constant periods over which the query is evaluated. While performance tradeoffs clearly depend on the underlying XQuery engine, we argue that there are queries that favor each of the five approaches.

1 Introduction

XML is now the emerging standard for data representation and exchange on the web. Querying XML data has garnered increasing attention from database researchers. XQuery [25] is the XML query language proposed by the World Wide Web Consortium. Although the XQuery working draft is still under development, several dozen demos and prototypes of XQuery processors can be found on the web. The major DBMS vendors, including Oracle [19], IBM [13], and Microsoft [17], have all released early implementations of XQuery.

Almost every database application involves the management of temporal data. Similarly, XML data changes over time with the creation, modification, and deletion of the XML documents. These changes involve two temporal dimensions, *valid time* and *transaction time* [20]. While there has been some work addressing the transaction time dimension of XML data [6, 7, 16], less attention has been focused on the valid time dimension of XML data. Expressing queries on temporal data is harder than writing queries on nontemporal data.

In this paper, we present a temporal XML query language, τXQuery, in which we add temporal support to XQuery by extending its syntax and semantics. Our goal is to move the complexity of handling time from the user/application code into the τXQuery processor. Moreover, we do not want to design a brand new query language. Instead, we made minimal changes to XQuery. Although we discuss valid time in this paper, the approach also applies to transaction time queries.

τXQuery utilizes the data model of XQuery. The few reserved words added to XQuery indicate three different kinds of valid time queries. *Representational queries* have the same semantics with XQuery, ensuring that τXQuery is upward compatible with XQuery. New syntax for *current* and *sequenced queries* makes these queries easier to write. We carefully made τXQuery compatible with XQuery to ensure the smooth migration from nontemporal application to temporal application; this compatibility also simplifies the semantics and its implementation.

To implement τXQuery, we adopt the stratum approach, in which a stratum accepts τXQuery expressions and maps each to a semantically equivalent XQuery expression. This XQuery expression is passed to an XQuery engine. Once the XQuery engine obtains the result, the stratum possibly performs some additional processing and returns the result to the user. The advantage of this approach is that we can exploit the existing techniques in an XQuery engine such as the query optimization and query evaluation. The stratum

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003

approach does not depend on a particular XQuery engine.

The paper focuses on how to perform this mapping, in particular, on mapping *sequenced* queries, which are by far the most challenging. The central issue of supporting sequenced queries (in any query language) is time-slicing the input data while retaining period timestamping. Timestamps are distributed throughout an XML document, rather than uniformly in tuples, complicating the temporal slicing while also providing opportunities for optimization. Any implementation of temporal support in a query language must come to terms with temporal slicing. This is the first paper to do so for XML.

The rest of the paper is organized as follows. We first present an example that illustrates the benefit of temporal support within the XQuery language. Temporal XML data is briefly introduced in Section 3. Section 4 describes the syntax and semantics of τXQuery informally. The following section provides a formal semantics of the language expressed as a source-to-source mapping in the style of denotational semantics. We then discuss the details of a stratum to implement τXQuery on top of a system supporting conventional XQuery.

The formal semantics utilizes *maximally-fragmented time-slicing*. Section 7 considers four optimizations: selected node time-slicing, copy-based per-expression time-slicing, in-place per-expression time-slicing, and idiomatic time-slicing. The related work is discussed in Section 8. Section 9 concludes the paper and lists interesting issues that are worthy of further study.

2 An Example

An XML document is static data; there is no explicit semantics of time. But often XML documents contain time-varying data. Consider *customer relationship management*, or *CRM*. Companies are realizing that it is much more expensive to acquire new customers than to keep existing ones. To ensure that customers remain loyal, the company needs to develop a relationship with that customer over time, and to tailor its interactions with each customer [2, 10]. An important task is to collect and analyze historical information on customer interactions. As Ahlert emphasizes, “It is necessary for an organization to develop a common strategy for the management and use of all customer information” [1], termed *enterprise customer management*. This requires communicating information on past interactions (whether by phone, email, or web) to those who interact directly with the customer (the “front desk”) and those who analyze these interactions (the “back desk”) for product development, direct marketing campaigns, incentive program design, and refining the web interface. Given the disparate software applications and databases used by the different departments in the company, using XML to pass this important information around is an obvious choice.

Figure 1 illustrates a small (and quite simplified) portion of such a document. This document would contain information on each customer, including the identity of the

```
<CRMdata>
  <customer supportLevel = "platinum">
     <contactInfo> ... </contactInfo>
     <directedPromotion> ... </directedPromotion>
     <supportIncident>
       <product>...</product>
       <description>...</description>
       <action>
         <who> ... <who>
         <what> ... <what>
         <handoff> ... <handoff>
       </action>
       <resolution> ...</resolution>
     </supportIncident>
     ...
  </customer>
...
</CRMdata>
```

Figure 1: A CRM XML document

customer (name or email address or internal customer number), contact information (address, phone number, etc.), the support level of the customer (e.g., silver, gold, and platinum, for increasingly valuable customers), information on promotions directed at that customer, and information on *support incidents*, where the customer contacted the company with a complaint that was resolved (or is still open).

While almost all of this information varies over time, for only some elements is the history useful and should be recorded in the XML document. Certainly the history of the support level is important, to see for example how customers go up or down in their support level. A support incident is explicitly temporal: it is opened by customer action and closed by an action of a staff member that resolves the incident, and so is associated with the period during which it is open. A support incident may involve one or several actions, each of which is invoked either by the original customer contact or by a hand-off from a previous action, and is terminated when a hand-off is made to another staff or when the incident is resolved; hence, actions are also associated with valid periods.

We need a way to represent this time information. In next section, we will describe a means of adding time to an XML schema to realize a *representational schema*, which is itself a correct XSchema [24], though we’ll argue that the details are peripheral to the focus of this paper. Instead, we just show a sliver of the time-varying CRM XML document in Figure 2. In this particular temporal XML document, a time-varying attribute is *represented* as a `timeVaryingAttribute` element, and that a time-varying element is represented with one or more *versions*, each containing one or more `timestamp` sub-elements. The valid-time period is represented with the beginning and ending instants, in a closed-open representation. Hence, the “gold” attribute value is valid for the day September 19 through the day March 19; March 19 is not included. (Apparently, a support level applies for six months.) Also, the valid period of an ancestor element (e.g., `supportIncident`) must contain the period(s) of descendant elements (e.g., `action`).

```
<CRMdata>
  <customer>
    <timeVaryingAttribute name="supportLevel"
       value="gold" vtBegin="2001-9-19"
       vtEnd="2002-3-19"/>
    <timeVaryingAttribute name="supportLevel"
       value="platinum" vtBegin="2002-3-19"
       vtEnd="2003-9-19"/>
     ...
    <supportIncident>
      <timestamp vtBegin="2002-4-11"
                 vtEnd="2002-4-29"/>
       ...
      <action>
        <timestamp vtBegin="2002-4-11"
                   vtEnd="2002-4-21"/>
        <who> ... </who> ...
      </action>
      <action> <timestamp .../>...</action>
    </supportIncident>
  </customer>
</CRMdata>
```

Figure 2: A temporal XML document for CRM

Note, though, that there is no such requirement between siblings, such as different `supportLevels` or between time-varying elements and attributes of an element.

Consider now an XQuery query on the static instance in Figure 1, "What is the average number of open support incidents per gold customer?" This is easily expressed in XQuery as

```
avg(for $c in document("CRM.xml")//
           customer[@supportLevel="gold"]
    return count($c/supportIncident))
```

Now, if the analyst wants the history of the average number of open support incidents per gold customer (which hopefully is trending down), the query becomes much more complex, because both elements and attributes are time-varying. (The reader is invited to try to express this in XQuery, an exercise which will clearly show why a temporal extension is needed.)

An XML query language that supports temporal queries is needed to fill the gap between XQuery and temporal applications. As we will see, this temporal query (the history of the average) is straightforward to express in τXQuery.

3 Temporal XML Data

The conventional schema defines the structure of the non-temporal data, which are simply XML instance documents. A time-varying XML document can be conceptualized as a series of conventional documents, all described by the same schema, each with an associated valid and/or transaction time. Hence we may have a version on Monday, the same version on Tuesday, a slightly modified version on Wednesday, and a further modified version on Thursday that is also valid on Friday. This sequence of conventional documents in concert comprise a single time-varying XML document.

The data designer specifies with a separate representational schema where in the time-varying document the timestamps should be placed, which is independent from which components in the document can change over time. For example, the user may want to add timestamps to a parent node if all sub-elements of that parent node are time-varying. An alternative design is to add timestamps to all the sub-elements. This is a desirable flexibility provided to the user. However, note that timestamps can occur at any level of the XML document hierarchy. τXQuery has to contend with this variability.

We emphasize that the representational schema is a conventional XML schema. The non-temporal schema for our CRM example would describe e.g., `customer` and `supportIncident` elements; the representational schema would add (for the document in Figure 2) the `timestamp` and `timeVaryingAttribute` elements. The rest of this paper is largely independent of these representational details.

Constraints must be applied to the temporal XML documents to ensure the validity of the temporal XML documents. One important constraint is that the valid time boundaries of parent elements must encompass those of its child. Violating this constraint means at some time, a child element exists without a parent node, which never appears in a valid document. Another constraint is that an element without timestamps inherits the valid periods of its parent. These constraints are exploited in the optimizations that will be discussed in Section 7.

4 The Language

There are three kinds of temporal queries supported in τXQuery: current queries, sequenced queries, and representational queries. We will introduce these queries and show an example of each kind of query. The next section provides the formal semantics for these queries, via a mapping to XQuery.

4.1 Current Queries

An XML document without temporal data records the current state of some aspect of the real world. After the temporal dimension is added, the history is preserved in the document. Conceptually, a temporal XML document denotes a sequence of conventional XML documents, each of which records a snapshot of the temporal XML document at a particular time. A current query simply asks for the information about the current state. An example is, "what is the average number of (currently) open support incidents per (current) gold customer?"

```
current avg(for $c in document("CRM.xml")//
            customer[@supportLevel="gold"]
            return count($c/supportIncident))
```

The semantics of a current query is exactly the same as the semantics of the XQuery (without the reserved word `current`) applied to the current state of the XML document(s) mentioned in the query. Applied to the instance in

Figure 2, that particular customer would not contribute to this average, because the support level of that customer is currently platinum.

Note that to write current queries, users do not have to know the representation of the temporal data, or even which elements or attributes are time-varying. Users can instead refer solely to the nontemporal schema when expressing current queries.

4.2 Sequenced Queries

Sequenced queries are applied independently at each point in time. An example is, "what is the history of the average number of open support incidents per gold customer?"

```
validtime avg(for $c in document("CRM.xml")//
           customer[@supportLevel="gold"]
           return count($c/supportIncident))
```

The result will be a sequence of time-varying elements, in this case of the following form.

```
<timeVaryingValue>
  <timestamp vtBegin="2001-1-1"
             vtEnd="2001-2-10"/>
  <value>4</value>
</timeVaryingValue>
<timeVaryingValue>
  <timestamp vtBegin="2001-2-10"
             vtEnd="2001-5-6"/>
  <value>2</value>
</timeVaryingValue>
...
```

Our CRM customer in Figure 2 would contribute to several of the values. As with current queries, users can write sequenced queries solely with reference to the nontemporal schema, without concern for the representation of the temporal data.

4.3 Representational Queries

There are some queries that cannot be expressed as current or sequenced queries. To evaluate these queries, more than one state of the input XML documents needs to be examined. These queries are more complex than sequenced queries. To write such queries, users have to know the representation of the timestamps (including time-varying attributes) and treat the timestamp as a common element or attribute. Hence, we call these queries representational queries. There is no syntactic extension for representational queries. An example is, "what is the average number of support incidents, now or in the past, per gold customer, now or in the past?"

```
avg(for $c in document("CRM.xml")//customer
    where $c/timeVaryingAttribute
     [@value="gold"][@name="supportLevel"]
    return count($c/supportIncident))
```

Such queries treat the `timeVaryingAttribute` and `timestamp` elements as normal elements, without any special semantics. Our customer in Figure 2 would participate in this query because she was once a gold member.

Representational queries are important not only because they allow the users to have full control of the timestamps, but also because they provide upward compatibility; any existing XQuery expression is evaluated in τXQuery with the same semantics as in XQuery.

5 Semantics

We now define the formal syntax and semantics of τXQuery statements, the latter as a source-to-source mapping from τXQuery to XQuery. We use a syntax-directed denotational semantics style formalism [23].

There are several ways to map τXQuery expressions into XQuery expressions. We show the simplest of them in this section to provide a formal semantics; we will discuss more efficient alternatives in Section 7. The goal here is to utilize the conventional XQuery semantics as much as possible. As we will see, a complete syntax and semantics can be given in just two pages by exploiting the syntax and semantics of conventional XQuery.

The BNF of XQuery we utilize is from a recent working draft [26] of W3C. The grammar of τXQuery begins with the following production. Note that the parentheses and vertical bars in an *italic* font are the symbols used by the BNF. Terminal symbols are given in a `sans serif` font.

A τXQuery expression has an optional modifier; the syntax of $\langle$Q$\rangle$ is identical to that of XQuery.

$\langle$TQ$\rangle$ ::=
(`current` | `validtime` (`[`$\langle$BT$\rangle$`,`$\langle$ET$\rangle$`]`)$^?$)$^?$ $\langle$Q$\rangle$

The semantics of $\langle$TQ$\rangle$ is expressed with the semantic function $\tau XQuery[\![\ ]\!]$, taking one parameter, a τXQuery query, which is simply a string. The domain of the semantic function is the set of syntactically valid τXQuery queries, while the range is the set of syntactically correct XQuery queries. The mapping we present will result in a semantically correct XQuery query if the input is a semantically correct τXQuery query.

5.1 Current Queries

The mapping of current queries to XQuery is pretty simple. Following the conceptual semantics of current queries, the current snapshot of the XML documents are computed first. Then, the corresponding XQuery expression is evaluated on the current snapshot.

$\tau XQuery[\![$`current` $\langle$Q$\rangle]\!] = cur[\![\langle$Q$\rangle]\!]$

$\langle$Q$\rangle$ ::= $\langle$QueryProlog$\rangle$ $\langle$QueryBody$\rangle$

$cur[\![\langle$Q$\rangle]\!]$ =

```
 import schema namespace rs=
  "http://www.cs.arizona.edu/tau/RXSchema"
```

```
 at "RXSchema.xsd"
declare namespace tau=
      "www.cs.arizona.edu/tau/Func"
snapshot [[⟨QueryProlog⟩]] current-dateTime()
define function tau:snapshot...
snapshot [[⟨QueryBody⟩]] current-dateTime()
```

The two namespaces defined in the above code are used by the auxiliary functions. `RXSchema.xsd` contains definitions of the `timestamp` and `timeVaryingAttribute` elements. The other namespace `tau` is defined for the semantic mapping. All the auxiliary functions and variables used for the mapping have this prefix. We use a new semantic function *snapshot* ⟦ ⟧ $w$hich takes an additional parameter, an XQuery expression that evaluates to the `xs:dateTime` type. As with other semantic functions utilized here, the domain is a τXQuery expression (a string) and the range is an XQuery expression (also a string).

In both ⟨QueryProlog⟩ (that is, the user-defined functions) and ⟨QueryBody⟩, only the function calls `document()` and `input()` need to be mapped. The rest of the syntax is simply retained. We show the mapping of `document()` below. A similar mapping applies to `input()`.

```
snapshot [[document(⟨String⟩)]] t =
 tau:snapshot(document(⟨String⟩), t)
```

The auxiliary function `snapshot()` (given elsewhere [11]) takes a node n and a time t as the input parameters and returns the snapshot of n at time t, in which the valid timestamps and elements not valid now have been stripped out.

5.2 Representational Queries

The mapping for representational queries is trivial.

τ*XQuery* ⟦⟨Q⟩⟧ = ⟨Q⟩

This mapping obviously ensures that τXQuery is upward compatible with XQuery.

5.3 Sequenced Queries

In a sequenced query, the reserved word `validtime` is followed by an optional period represented by two dateTime values enclosed by a pair of brackets. If the period is specified, the query result contains only the data valid in this period. The semantics of sequenced queries utilizes the *seq* ⟦ ⟧ semantic function, which we will provide shortly.

```
τXQuery [[validtime ⟨Q⟩]] =
  seq [[⟨Q⟩]] $tau:period("1000-01-01",
                          "9999-12-31")
```

When there is no valid-time period specified in the query, the query is evaluated in the whole timeline the system can represent. If the valid-time period is explicitly specified by the user, the translation is as follows.

```
τXQuery [[validtime [⟨BT⟩,⟨ET⟩] ⟨Q⟩]] =
 seq [[⟨Q⟩]] tau:period(⟨BT⟩,⟨ET⟩)
```

As with *snapshot* ⟦ ⟧, the sequenced semantic function *seq* ⟦ ⟧ has a parameter, in this case an XQuery expression that evaluates to an XML element of the type `rs:vtExtent`. This element represents the period in which the input query is evaluated.

The semantics of a sequenced query is that of applying the associated XQuery expression simultaneously to each state of the XML document(s), and then combining the results back into a period-stamped representation. We adopt a straightforward approach to map a sequenced query to XQuery, based on the following simple observation first made when the semantics of temporal aggregates were defined [21]: the result changes only at those time points that begin or end a valid-time period of the time-varying data. Hence, we can compute the *constant periods*, those periods over which the result is unchanged. To compute the constant periods, all the timestamps in the input documents are collected and the begin time and end time of each timestamp are put into a list. These time points are the only modification points of the documents, and thus, of the result. Therefore, the XQuery expression only needs to be evaluated on each snapshot of the documents at each modification point. Finally, the corresponding timestamps are added to the results.

```
seq [[⟨Q⟩]] p =
 import schema namespace rs=
  "http://www.cs.arizona.edu/tau/RXSchema"
   at "RXSchema.xsd"
 import schema namespace tvv=
  "http://www.cs.arizona.edu/tau/Tvv"
   at "TimeVaryingValue.xsd"
 declare namespace tau=
  "www.cs.arizona.edu/tau/Func"
 seq [[⟨QueryProlog⟩]] p
define function tau:all-const-periods...
...
 for $tau:p in
   tau:all-const-periods(p, getdoc [[⟨Q⟩]] )
 return tau:associate-timestamp($tau:p,
      timeslice [[⟨QueryBody⟩]] $tau:p/@vtBegin)
```

The namespace `tvv` defines the sequenced time-varying value type needed in the mapping. The schema that defines `tvv` is given elsewhere [11]. *getdoc* ⟦ ⟧ takes a query string as input and returns a string consisting of a parenthesized, comma-separated list of the function calls of `document()` that appear in the input string, along with those mentioned in the definitions of functions invoked by the input string.

The function `all-const-periods()` takes this list of document nodes as well as a time period (represented as two `dateTime` values) and computes all the periods during which no single value in any of the documents changes. The returned periods should be contained in the input period, specified by the first parameter. This function first

finds all the closed-open time points in all the input documents and contained in the input period. Then it sorts this list of time points and removes duplicates. The period between each pair of points that are adjacent forms a [closed–open) constant period. For example, if three time points 1, 3, and 5 are found, then a list of two `timestamp` elements representing the periods [1–3) and [3–5) is returned. The input documents and the result are all constant over each of these periods.

The function `associate-timestamp()` takes a sequence of items and a `timestamp` element as input and associates the timestamp representing the input period with each item in the input sequence. Both this and the previous function are auxiliary functions that depend on the representation. Again, the definitions are provided elsewhere [11], for the particular representation in Figure 2.

We need to *time-slice* all the documents on each of the constant periods computed by the auxiliary function `all-const-periods()` and evaluate the query in each time slice of the documents (in Section 7, we examine more sophisticated slicing strategies). Since the documents appearing in both ⟨QueryProlog⟩ and ⟨QueryBody⟩ need to be time-sliced, we define *seq* ⟦⟨QueryProlog⟩⟧ p and *timeslice* ⟦⟨QueryBody⟩⟧ t further. In ⟨QueryProlog⟩, only the function definitions need to be mapped. We add an additional parameter (a time point) to each user-defined function and use this time point to slice the document specified in the function.

⟨FunctionDefn⟩ ::= `define function` ⟨FuncName⟩
`(`⟨ParamList⟩$^?$`)returns` ⟨SequenceType⟩
`{`⟨ExprSequence⟩`}`

seq ⟦⟨FunctionDefn⟩⟧ p =
`define function` ⟨FuncName⟩`(xs:dateTime`
`$tau:time,` ⟨ParamList⟩$^?$`) returns` ⟨SequenceType⟩
`{`*timeslice* ⟦⟨ExprSequence⟩⟧ `$tau:time}`

In ⟨ExprSequence⟩, only the function calls need to be changed. The functions are partitioned into two categories: the user-defined functions and the built-in functions. All the user-defined functions have one more parameter, therefore calling the functions should be changed accordingly.

⟨FunctionCall⟩ ::=
⟨QName⟩`(` `(`⟨Expr⟩ `(` `,` ⟨Expr⟩`)`* `)`$^?$ `)`

For user-defined functions, the semantics is defined as follows.

timeslice ⟦⟨FunctionCall⟩⟧ t =
⟨QName⟩`(`t`,` `(`*timeslice* ⟦⟨Expr⟩⟧ t
`(`, *timeslice* ⟦⟨Expr⟩⟧ t`)`*`)`$^?$`)`

The function `document()` is the only built-in function that needs to be mapped.

timeslice ⟦`document(`⟨String⟩`)`⟧ t =
`tau:snapshot(document(`⟨String⟩`),` t`)`

⟨QueryBody⟩ is actually an ⟨ExprSequence⟩. We will not repeat the above mapping for ⟨QueryBody⟩. Note that the function call `input()` is treated the same as the function call `document()`, in that it should also be time-sliced. For the brevity, we do not show that mapping here.

Time-slicing a document on a constant period is implemented by computing the snapshot of the document at the begin point of the period. There are two reasons that we add one more parameter to user-defined functions and introduce a new function *timeslice* ⟦ ⟧ instead of using the existing function *snapshot* ⟦ ⟧. First, the constant periods are computed in XQuery, but the query prolog must proceed the query body which includes the computation of the constant periods. Secondly, at translation time it is not known on which periods the documents appearing inside function definitions should be time-sliced. This is not a problem for current queries, where it is known when (now) the snapshot is to be taken.

The result of a sequenced query should have the valid timestamp associated with it, which is not the case for a conventional XQuery expression. Thus, the type of the result from a sequenced statement is different from that from a representational or current statement. The XQuery data types are mapped to timestamped types by `associate-timestamp()`. A single value of an atomic type is mapped to a sequence of elements with the type `tvv:timeVaryingValueType`, as shown in the example in Section 4.2. The mapping of other XQuery data types is given elsewhere [11].

One concern is how to maintain the order of values within sequences. Queries can be divided into three broad classes regarding the order of the result. The first class consists of queries that do not care about the order. Any order that is returned is fine. The second class consists of queries that explicitly sort the resulting sequence, via the XQuery `sortby` operator. In our mapping, the sequences are sorted on the constant period, using a stable sort to retain the order within a constant period, and then timestamped and concatenated. This ensures that the timeslice of this sequence at any point in time would result in the correct order. The third class contains queries that do not have a `sortby` operator yet is not an unordered query. Here according to the way that the result is sorted, with a stable sort by the begin time of the constant period, the document order of the sequence in each constant period is retained. Thus, for all the three classes, the order of the result sequence is correct.

5.4 Summary

There are three modes in τXQuery. Representational queries are syntactically and semantically identical to XQuery queries. Current queries are evaluated on a snapshot of each time-varying document. As the snapshot will contain no `timestamp` nor `timeVaryingAttribute` elements, the conventional XQuery semantics can be used.

Interestingly, for sequenced queries, once the document(s) are timesliced based on the constant periods, we

can again utilize the conventional XQuery semantics, thus ensuring *snapshot reducibility* [14, 22]. Effectively, a sequenced query is treated as a series of conventional queries, based on the constant periods. This provides a pleasing symmetry in the formal semantics of the three modes.

Our approach is independent of the representation (other than the details of some of the XQuery functions utilized by the mapping); in particular, it is independent of the location of the timestamps within the document.

6 Stratum Architecture

We would like to carry over the nice symmetry of the semantics into the implementation of τXQuery. We do so by utilizing a *stratum* approach. Each τXQuery expression is mapped to an XQuery expression, which is passed to an XQuery processor for evaluation.

The architecture of the τXQuery stratum is shown in Figure 3. The dashed rectangle indicates the boundary of the stratum. When a query is input, the initial keyword is examined to determine the kind of query. A representational query is passed to the underlying XQuery processor directly, while a current or sequenced query must be converted by the appropriate mapper to effect the translation given in Section 5. The resulting XQuery expression is sent to the XQuery processor.

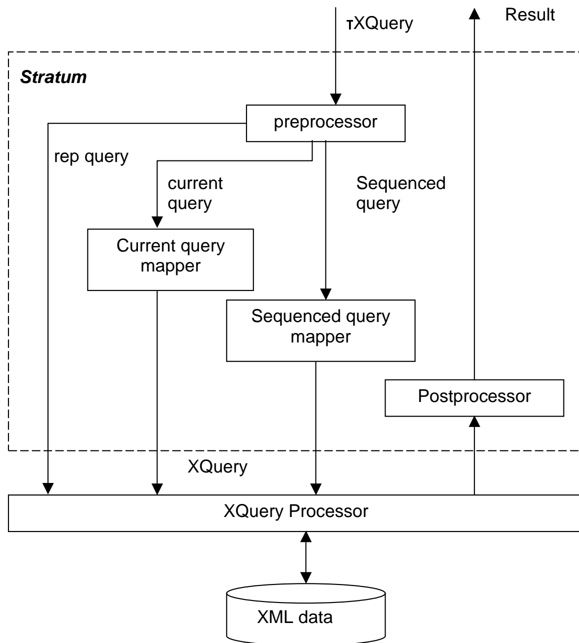


Figure 3: Architecture of the τXQuery stratum

The two mappings are straightforward. One interesting aspect is that all the semantic functions mentioned in the previous section are implemented directly in the query mappers in the stratum. For example, the *getdoc* ⟦ ⟧ semantic function discussed briefly in Section 5.3 is implemented by the sequenced query mapper. The documents mentioned in the query (and in functions called directly or indirectly by the query) can be determined from a syntactic analysis of the query except when the document name is a computed string (this situation is discussed in Section 7.6); no interaction with the XQuery processor is required for that semantic function. The other semantic functions are also evaluated in the mappers, to convert a τXQuery expression as a text string into an XQuery expression, again as a text string.

Once the XQuery processor has evaluated the query, the stratum's postprocessor coalesces the query results. *Coalescing* in relational temporal databases is a unary operator [3, 8, 14]; it reduces the number of tuples by eliminating duplicate values valid at the same time and merge tuples that have adjacent time periods and that agree on the explicit attribute values. Coalescing in an XML context involves merging versions of elements that have identical subelements and whose periods of validity are adjacent.

Of the three kinds of queries discussed in Section 4, current queries do not return a time-varying result, and so coalescing is not relevant. For representational queries, we do not (and indeed cannot) coalesce the result. Hence, coalescing is only relevant for sequenced queries.

In most cases, the result of a sequenced query is a sequence of elements. Associated with each element is a timestamp, denoting some period of time. This period is a constant period of its parent element. However, it may not be the maximal constant period of its parent element. Consider the example query used in Section 4.2. It is possible that the average is 5 during two separate but adjacent periods. In this case, the result is uncoalesced (the result is represented by two elements when one would do). Coalescing this result will merge the two elements into one.

Coalescing temporal XML data is different in many aspects from coalescing relational data. It is an open question whether coalescing can be done efficiently in XQuery, or whether this computation is best done in the stratum.

7 Optimization of Slicing

In Section 5.3, we presented one method to map sequenced τXQuery expressions to XQuery. In this method, we time-sliced all the input documents at the finest granularity of modification time by using every single time point present as a begin time or an end time in a `timestamp` or `timeVaryingAttribute` element contained in each document. We call this method *maximally-fragmented time-slicing*. (We emphasize that this approach is far more efficient than taking a timestamp of the document at every time point in which it is valid, termed *unfolding* in the context of temporal relations [18]. Maximally-fragmenting still uses the periods in the data to compute the constant periods.) However, a query may not touch the information of the most frequently updated elements.

In the CRM example in Figure 2, the most frequently changing element is `action`. Maximally-fragmented time-slicing always slices the document on the constant periods of `action`. The example query in Section 4.2 does not go all the way down to `action`. In particular, examining Figure 2 indicates that a constant period of [2002-4-11–2002-4-29) is sufficient, without being broken

into two periods at 2002-4-21. Slicing the whole document at all the time points found in the timestamp periods often involves too much work over too many constant periods. In this section, we discuss several optimizations that compute fewer constant periods and slice only portions of the document; these optimizations are largely independent of the query language and representation. Details of the semantic mapping for these optimizations are given elsewhere [11].

7.1 Selected Node Slicing

Given a query string, the stratum can find all the names of the elements and the attributes specified in the query. Collecting the valid time points of only these nodes, constructing the constant periods for them, and time-slicing the documents only on these constant periods is sufficient. Each of the constant periods found in this process is the coarsest period during which all the nodes specified in the query are guaranteed to be stable. In this way, the query body is evaluated in fewer periods in the generated XQuery. Thus, the translated query is expected to be more efficient. An added benefit is that the result may already be coalesced, without further effort by the stratum.

For the example query mentioned in the last section, the stratum first determines that the elements specified in the query are `customer` and `supportIncident`; the time-varying attribute `supportLevel` is also referenced. The XQuery function `element-const-periods()` takes a sequence of documents and a sequence of strings as node names (elements or attributes) to collect the times appearing at those nodes (or inherited from ancestor nodes, if not timestamped directly) and then constructs the constant periods. If the schema is available, the stratum can instruct this function as to when to stop descending through the XML data, via a third parameter.

This method and the maximally-fragmented time-slicing method both time-slice the documents at the document level on a sequence of constant periods. However, a query may not touch a large part of the document. Time-slicing this untouched part is wasted work. In the next two sections, we present methods that avoid time-slicing the unused subtrees.

7.2 Per-Expression Slicing

XQuery is a functional language which allows various kinds of expressions to be nested with full generality. *Per-expression slicing* time-slices the subtree that is referenced by the relevant portion of the recursively evaluated query expression; this slicing is only on the constant periods of the root of this subtree. The sequenced version of the current expression then is evaluated on the time-sliced subtree. The result, a sequence of trees each of which associated with valid time-stamps, is again time-sliced on the constant periods of these trees for the evaluation of the expression at the next level. The constant periods in the subsequent level are shorter than, and contained within, the constant periods in the previous level. Thus, those unused subtrees are pruned before they are time-sliced. Since some of the

```
validtime avg(for $c in
 (let $tau:sequence:=document("CRM.xml")
  return
    for $tau:dot in $tau:sequence return
     $tau:dot/descendant-or-self::customer)
              return
      count(let $tau:sequence := $c return
       for $tau:dot in $tau:sequence return
         $tau:dot/child::supportIncident))
```

Figure 4: Normalizing the example query

nodes do not have timestamps, we need a way to remember the valid period for such nodes. In the section, we will present two per-expression slicing approaches: copy-based and in-place per-expression slicing. They utilize different methods to record the valid periods for the intermediate results.

7.3 Copy-Based Per-Expression Slicing

To record the valid periods of the intermediate results, copy-based slicing timestamps all the intermediate results no matter whether they are timestamped in the original document. During the query evaluation, copy-based slicing prunes the irrelevant portion of the document tree either because that portion is not referenced in the query or because that portion is not valid in the input period. This pruning is done by copying the relevant portion and then associating every element and attribute with the exact timestamp.

The stratum maps each non-terminal in a parsed τXQuery expression to a segment of valid XQuery code. Each production is handled individually, to minimize the slicing that is required. Since any XQuery program can be normalized by using the core grammar [26] (a subset of the XQuery grammar) provided by the W3C, defining the semantics on just the core grammar of τXQuery is sufficient. Rather than listing the complete translation rule for each production, which is tedious and is provided elsewhere [11], we use an example to illustrate the mapping.

Consider the example query "what is the history of the average number of open support incidents per customer?"

```
validtime avg(for $c in
        document("CRM.xml")//customer
      return count($c/supportIncident))
```

The normalized result of this query is shown in Figure 4. This result is obtained by applying the normalization formally defined in W3C working draft [26]. The only difference is we change the prefix `fs` to `tau`, since the normalization is the starting point of per-expression slicing and is treated as part of the mapping. We do not normalize built-in functions. Each step of a path expression is converted to some `let` and `for` expressions. The length of the query is increased while the number of distinct nonterminals to be dealt with is reduced. Some complicated expressions such as FLWR expressions and quantified expressions are removed during normalization.

Normalization is performed within the stratum before the translation of sequenced queries. The mapping is defined by the function $cb[\![\,]\!]\,p$. The period p is propagated

from the top level of the expression to the bottom during the mapping. The normalized result of this example includes four kinds of non-terminals of the core grammar: ⟨FunctionCall⟩, ⟨ForExpr⟩, ⟨LetExpr⟩, and ⟨PathExpr⟩. Each of the four kinds is time-sliced individually.

The outermost part of the example query is a function call which calls the built-in function `avg()`. Copy-based slicing maps the built-in function calls by going through the following steps. First, all the constant periods of the input data (and the subtree rooted at the input data) are found and put into a sorted sequence. Then, the original function is called once on each snapshot of the input data on each constant period. Finally the results are timestamped accordingly.

We have seen the production of ⟨FunctionCall⟩ in Section 5.3. The mapping of it is defined as follows.

```
cb ⟦⟨FunctionCall⟩⟧ p =
  let $tau:par1 := cb ⟦⟨Expr1⟩⟧ p
  let $tau:par2 := cb ⟦⟨Expr2⟩⟧ p
  for $tau:p in tau:all-const-periods(p,
          union($tau:par1, $tau:par2))
  return tau:associate-timestamp($tau:p,
      ⟨QName⟩(tau:snapshot($tau:par1,
                     $tau:p/@vtBegin),
              tau:snapshot($tau:par2,
                     $tau:p/@vtBegin)))
```

The query in Figure 4 calls the function `avg()`, which has only one input parameter. In this particular query, the ⟨$Expr_1$⟩ in the first `let` expression is replaced with a ⟨ForExpr⟩, and the second `let` expression is redundant and thus removed in the mapping.

The syntax production and the translation of ⟨ForExpr⟩ are defined as follows.

```
⟨ForExpr⟩ ::=
 (for $⟨VarName⟩ in ⟨Expr⟩ return)*
 ⟨TypeswitchExpr⟩

cb ⟦⟨ForExpr⟩⟧ p =
  for $tau:i in cb ⟦⟨Expr⟩⟧ p
  for $tau:p in tau:periods-of($tau:i)
  let $⟨VarName⟩ :=
  tau:copyrestrictedsubtree($tau:p,$tau:i)
  return cb ⟦⟨TypeswitchExpr⟩⟧ $tau:p
```

The auxiliary function `periods-of()` returns all the timestamps associated with the input node. The function `copyrestrictedsubtree()` takes one or more time periods and a variable as inputs. It propagates the time period from the top node of the variable to all its descendants by removing the branches not valid in the input periods and modifying the timestamps of the remaining nodes.

We continue the translation of the example query by replacing the ⟨Expr⟩ and ⟨TypeswitchExpr⟩ in the definition above with ⟨LetExpr⟩ and ⟨FunctionCall⟩ respectively. In XQuery, ⟨LetExpr⟩ binds a variable to the value of an expression which could be a single item or a sequence. In sequenced τXQuery, the expression is evaluated to a sequence even it is a single item at each time point. So the expression is time-sliced in each constant period to ensure the variable is bound to the correct value. The syntax production and mapping of ⟨LetExpr⟩ is as follows.

```
⟨LetExpr⟩ ::=
 (let $⟨VarName⟩ := ⟨Expr⟩ return)*
 ⟨TypeswitchExpr⟩

cb ⟦⟨LetExpr⟩⟧ p =
  let $tau:s :=cb ⟦⟨Expr⟩⟧ p
  for $tau:p in
        tau:const-periods(p, $tau:s)
  let $⟨VarName⟩ :=
  tau:copy-restricted-subtree($tau:p,
                              $tau:s)
  return cb ⟦⟨TypeswitchExpr⟩⟧ $tau:p
```

The function `const-periods()` is similar to `all-const-periods()`. The only difference is that the former returns the constant periods for each of the nodes in the input sequence, not for all the subelements.

In the example query, the ⟨LetExpr⟩ returns another ⟨ForExpr⟩, which returns a ⟨PathExpr⟩. A ⟨PathExpr⟩ in the normalized query has only one step, as shown in the following production. The ⟨PathExpr⟩ in the example query is the first case, a ⟨ForwardStep⟩.

```
⟨PathExpr⟩ ::= $⟨VarName⟩/⟨ForwardStep⟩
  | $⟨VarName⟩/⟨ReverseStep⟩
  | ⟨PrimaryExpr⟩

⟨ForwardStep⟩ ::= ⟨ForwardAxis⟩ ⟨NodeTest⟩

cb ⟦$⟨VarName⟩/⟨ForwardStep⟩⟧ p =
  for $tau:step in
      $⟨VarName⟩/⟨ForwardAxis⟩ cb ⟦⟨NodeTest⟩⟧ p
  where not(tau:special-node($tau:step))
  return tau:copyrestrictedsubtree(p,
                              $tau:step)
```

In the example query, the ⟨ForwardAxis⟩ and the ⟨NodeTest⟩ are `descendant-or-self::` and `customer` respectively. The fuction *cb* ⟦⟨NodeTest⟩⟧ maps the namespace of the input node name to the corresponding timestamped namespace. Since all the elements and attributes are timestamped during evaluation, their structures are changed and do not belong to their original namespaces. The stratum defines a timestamped analog for each namespace. Whenever a namespace is referenced in the query, it is mapped to its timestamped analog. The function `special-node()` returns true when the input node is a special node (e.g., `timestamp` and `timeVaryingAttribute`) for representing the valid periods. This where clause filters out those special nodes when the ⟨NodeTest⟩ is a wildcard.

So far, we have seen how to translate the example sequenced query to XQuery. As this example does not cover

the whole language (the translation of the remainder of the core grammar in all of its glory may be found elsewhere [11]), there are some other interesting problems we would like to discuss here. Most of the expressions, though are not shown in this paper, are similar to ⟨LetExpr⟩ in that they first find the constant periods at the current level, then evaluate the following expressions in each constant period. Since time-varying attributes are represented by elements, the translation should take care of this conversion, which can be found in the long version of this paper. Due to the copy-based nature, the results at each step are not the original nodes in the documents, but copies of those nodes with the same value in the corresponding valid periods. It is easy to understand that the ancestor information cannot be obtained. Thus, this approach does not work for reverse axis in path expression, nor for some built-in funtions that need the identifier of the original node. In the next section, we introduce a per-expression slicing approach that works for the entire language.

7.4 In-Place Per-Expression Slicing

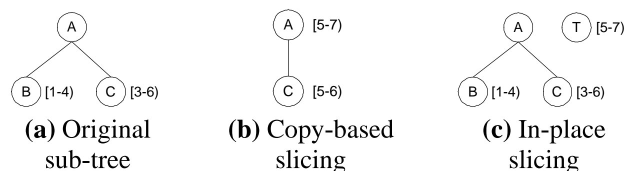
Rather than timestamping all the intermediate results, in-place per-expression slicing keeps all the intermediate results with the document. To record the valid period of these intermediate results, it puts the intermediate results and their actual timestamps in one sequence in the form of (item, timestamp, item, timestamp, ...). When the evaluation of the query is finished, the stratum associates the actual timestamps with each item to obtain the final result. In this way, the XQuery engine can identify each node in the context of the original document and find the ancestor of each node as well.

As an example, the ⟨ForExpr⟩ is translated as follows (cf., the definition of *cb* ⟦⟨ForExpr⟩⟧ *p* in the previous section).

```
inp ⟦⟨ForExpr⟩⟧ p =
  let $tau:s:=inp ⟦⟨Expr⟩⟧ p
  for $tau:v in $tau:s
  let $tau:vi:=index-of($tau:s, $tau:v)
  where ($tau:vi mod 2 = 1) return
    let $tau:p:=item-at($tau:s, $tau:vi+1)
    let $⟨VarName⟩ := ($tau:v,$tau:p)
    return inp ⟦⟨TypeswitchExpr⟩⟧ $tau:p
```

In this approach, whenever an item is needed in the evaluation, an (item, timestamp) pair is provided. The difference between copy-based slicing and in-place slicing is shown in Figure 5. Suppose, a sub-tree rooted at A without timestamps has two timestamped sub-elements B and C (Figure 5(a)). Suppose this sub-tree is the intermediate result for some evaluation on the period of [5-7), copy-based slicing makes a copy of the relevant portion with the correct timestamp as shown in Figure 5(b), while in-place slicing returns the original sub-tree with an actual timestamp as shown in Figure 5(c).

In-place slicing can handle all the sequenced queries in the cost of keeping more data in the intermediate results



(a) Original sub-tree **(b)** Copy-based slicing **(c)** In-place slicing

Figure 5: Intermediate results for per-expression slicing

and generating longer XQuery expressions. On the other hand, since it does not change the nodes in the intermediate results, the timestamped analog for each namespace and data type is not needed.

7.5 Idiomatic Slicing

Idiomatic slicing applies to copy-based per-expression slicing. As we have seen, the normalization of path expressions is tedious. A path expression with one step is normalized to at least three lines of let-for expressions. If there is a path expression with multiple steps, the result of the normalization will be much longer than the path expression. In each step, the data is time-sliced and the valid timestamps are propagated to the lower level nodes. Since `let` and `for` expressions both time-slice the expression appearing in them, there are a lot of time-slices generated. For example, the variable `$tau:sequence` is sliced at least twice in each step.

To avoid the extra slicing, a path expression can be translated without normalization. This is an instance of *idiomatic slicing*, in which two or more consecutive expressions in a query are analyzed as a unit to determine where the time-slicing most profitably should occur. We defines an auxiliary function `tau:seqpath()` to evaluate path expressions for copy-based slicing. This function works for most path expressions except for those with predicates.

There are several situations in which idiomatic slicing applies. One is when a `let` expression binds a variable `$a` to a sequence, followed by a `for` expression that binds a variable `$b` to each of the items in `$a`. When the `for` expression is translated, there is no need to evaluate `$a` in sequenced semantics, because the evaluation period for `$a` does not change and the function `copyrestricted-subtree()` will do useless work on `$a`.

7.6 Comparison

We have proposed five ways to effect time-slicing of the input documents into constant periods to enable sequenced queries. Maximally-fragmented time-slicing produces the shortest XQuery expressions. It works in all cases except where the name of a document is itself an expression. Selected node time-slicing reduces the number of constant periods, sometimes significantly, at the expense of more analysis by the stratum. Per-expression slicing reduces the number of constant periods further, while also not requiring the entire document to be sliced. It can handle the name of a document as an expression. Although copy-based slicing cannot handle reverse steps in path expressions nor a few built-in functions, in-place slicing supports the entire

language. One drawback of per-expression slicing is further analysis by the stratum, and expansion of a query into the core grammar. Idiomatic time-slicing, a refinement of copy-based slicing, may shorten both the resulting XQuery and the time complexity of that query by slicing more judiciously.

While performance tradeoffs clearly depend on the way in which the underlying XQuery engine implements conventional XQuery statements, we now show that there are queries and documents that favor each of the five approaches.

Maximally-fragmented slicing. Consider a document with every node timestamped with the same period. A query asks for all the sub-elements (specified as a wildcard) under a particular element over the entire timeline. Since there is only one constant period, maximally-fragmented slicing time-slices the document only at the beginning time and evaluates the query only once. Selected node slicing does not work due to the wildcard. Other slicing approaches need to propagate the timestamp at each level of the document, which is not necessary in this case.

Selected node slicing. Consider a document with every node timestamped. There is one element named `e` and all its ancestors and siblings have the same very long valid period, while its descendants have very short periods. A query asks for the element `e` favors this approach, because it time-slices the document only once. Maximally-fragmented slicing has to time-slice the document many times. Other approaches again need to propagate the timestamp from the root.

Copy-based per-expression slicing. Consider a document with some parent and its child elements timestamped. Each of the children has many versions. A query asks for the second child element in a short period, but not the shortest period in the document. Copy-based slicing filters out a large portion of the document tree early at upper level of the evaluation. Maximally-fragmented slicing and selected node slicing both slice the whole document on many short constant periods. In-place slicing keeps more sub-elements in the intermediate results. Idiomatic slicing does not work for the path expression with position predicates.

In-place per-expression slicing. Consider the same document as in the last paragraph. Now the query is changed to ask for the second child element that has an ancestor named `a` in a short period. Copy-based slicing cannot handle ancestors. Other approaches still have the disadvantages mentioned in the last paragraph.

Idiomatic slicing. Use the same document. When the query asks for all the child elements in a short period without position predicates, idiomatic slicing is best in that it reduces the size of the result XQuery code and it avoids repeatedly slicing some intermediate nodes.

8 Related Work

The related work includes the research of querying relational temporal databases and the more recent work on the temporal aspect of XML data.

As mentioned in Section 1, there has been some work addressing the transaction time dimension of XML data [6, 7, 16]. These papers focus on aspects of XML versioning, including representing, detecting, and querying the changes in XML documents. Our work concentrates on how to evaluate τXQuery by leveraging existing XQuery engines. The time-slicing approaches do not depend on the representation of the temporal information and they work fine for transaction time querying.

Dyreson et al. proposed a framework for capturing and querying meta-data properties including temporal information in a semistructured data model [9]. This work can be viewed as an extension to a conventional semistructured database. Temporal constituents in XML and their representation were investigated by Manukyan et al. [15]. They did not address the problem of querying temporal XML.

Grandi and Mandreoli [12] introduced valid time into the XML documents and an extension to XQL to express temporal predicates. In our terminology, their approach would be considered to support representational queries with additional predicates. Buneman et al. presented a timestamp-based approach to archive scientific data [4]. They focus on how to merge different versions (documents) to one document with some nodes timestamped. Their work may be helpful to temporal coalescing of XML data.

9 Summary and Future Work

In this paper, we have presented a temporal XML query language, τXQuery, that minimally extends the syntax and semantics of XQuery. This language supports three kinds of queries: current, sequenced, and representational. A stratum approach is used to exploit the presence of XQuery implementations. Time-slicing the documents on constant periods is the main technique used in the translation. We proposed five time-slicing methods to map current and sequenced τXQuery expressions to XQuery.

Our approaches work on both valid time and transaction time data and queries. They are independent of the representation (the dependencies appear only in the auxiliary XQuery functions).

Future work includes comparing the different time-slicing methods empirically and further optimizing the mappings to eliminate redundant XQuery code and constant periods, and to exploit the schema. How to efficiently coalesce temporal XML data is an open question. Also of interest are techniques to augment the underlying XQuery evaluation engine to more efficiently support costly τXQuery queries. In some applications, data stored in a relational database is published as XML data. Mapping τXQuery expressions to SQL given the correspondence between the relational schema and the XML schema would be useful.

Acknowledgments

We thank Bengu Li and Curtis Dyreson for help in the initial stages and Merrie Brucks and Shankar Ganesan of the

University of Arizona Department of Marketing for help with the CRM case study. This research was supported in part by NSF grants IIS-0100436 and EIA-0080123 and grants from the Boeing Corporation and Microsoft.

References

[1] H. Ahlert, "Enterprise Customer Management: Integrating Corporate and Customer Information," in **Relationship Marketing**, Springer, 2000.

[2] J. Anton and N. L. Petouhoff, **Customer Relationship Management**, Prentice Hall, 2002.

[3] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo, "Coalescing in Temporal Databases," in *Proceedings of the International Conference on Very Large Databases*, pp. 180–191. Bombay, India, September 1996.

[4] P. Buneman, S. Khanna, K. Tajima, and W-C. Tan, "Archiving Scientific Data," in *Proceedings of the ACM SIGMOD International Conference*, pp.1–12. Madison, Wisconsin, June 2002.

[5] S-Y. Chien, V. J. Tsotras, and C. Zaniolo, "Efficient Schemes for Managing Multiversion XML Documents," *the VLDB Journal*, Volume 11. Issue 4. (2002) pp. 332–353.

[6] S-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang, "Efficient Complex Query Support for Multiversion XML Documents," in *Proceedings of the International Conference on EDBT*, pp. 25-27. Prague, Czech, March 2002.

[7] G. Cobena, S. Abiteboul, and A. Marian, "Detecting Changes in XML Documents," in *Proceedings of the IEEE International Conference on Data Engineering*, pp. 41–52. San Jose, February 2002.

[8] C. E. Dyreson, "Temporal Coalescing with Now, Granularity, and Incomplete Information," in *Proceedings of the ACM SIGMOD International Conference*, San Diego, CA, June 2003.

[9] C. E. Dyreson, M. H. Bohlen, and C. S. Jensen, "Capturing and Querying Multiple Aspects of Semistructured Data," in *Proceedings of the International Conference on Very Large Databases*, pp. 290–301. Edinburgh, Scotland, 1999.

[10] S. Gallant, G. Piatetsky-Shapiro, and M. Tan, "Value-based Data Mining for CRM," in Proceedings of the SIGKDD International Conference, 2003.

[11] D. Gao and R. T. Snodgrass, "Syntax, Semantics, and Evaluation of the τXQuery Temporal XML Query Language," TimeCenter TR-72, March 2003. `www.cs.auc.dk/TimeCenter/pub.htm`

[12] F. Grandi and F. Mandreoli, "The Valid Web: A XML/XSL Infrastructure for Temporal Management of Web Documents," in *Proceedings of International Conference on Advances in Information Systems*, pp. 294-303. Izmir, Turkey, October 2000.

[13] IBM, "Xperanto Technology Demo," Mar 2002. `http://www7b.boulder.ibm.com/dmdd/library/demos/0203xperanto/0203xperanto.html`.

[14] C. S. Jensen and C. E. Dyreson (eds), "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), Springer-Verlag, pp. 367–405, 1998.

[15] M. G. Manukyan and L. A. Kalinichenko, "Temporal XML," in *Proceedings of ADBIS*, Vilnius, Lithuania, September 2001.

[16] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet, "Change-centric Management of Versions in an XML Warehouse," in *Proceedings of International Conference on Very Large Databases*, pp. 581-590. Rome, Italy, September 2001.

[17] Microsoft Corporation, "XML Query Language Demo," `http://131.107.228.20/xquerydemo`

[18] N. A. Lorentzos and Y. G. Mitsopoulos, "SQL Extension for Interval Data," *IEEE Transactions on Knowledge and Data Engineering* 9(3): 480–499, 1997.

[19] Oracle Corporation, "Oracle XQuery Prototype: Querying XML the XQuery way," March 2002. `http://otn.oracle.com/sample_code/tech/xml/xmldb/xmldb_xquerydownload.html`.

[20] R. T. Snodgrass and I. Ahn. "Temporal Databases". *IEEE Computer*, 19(9):35–42, September 1986.

[21] R. T. Snodgrass, S. Gomez, and L. E. McKenzie, "Aggregates in the Temporal Query Language TQuel". *IEEE Transactions on Knowledge and Data Engineering* 5(5): 826-842 (1993)

[22] R. T. Snodgrass, "The Temporal Query Language TQuel". *ACM Transactions on Database Systems* 12(2): 247-298 (1987)

[23] J. E. Stoy, **Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory**. The MIT Press 1979.

[24] World Wide Web Consortium, "XML Schema Part 0: Primer," *W3C Recommendation*, May, 2001. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502`

[25] World Wide Web Consortium, "XQuery 1.0: An XML Query Language," *W3C Working Draft*, August, 2002. `http://www.w3.org/TR/2002/WD-xquery-20020816/`

[26] World Wide Web Consortium, "XQuery 1.0 and XPath 2.0 Formal Semantics," *W3C Working Draft*, August, 2002. `http://www.w3.org/TR/2002/WD-query-semantics-20020816/`