Syntax, Semantics, and Query Evaluation in the τXQuery Temporal XML Query Language

Dengfeng Gao and Richard T. Snodgrass

March 16, 2003

TR-72

A TIMECENTER Technical Report

Title	Syntax, Semantics, and Query Evaluation in the τXQuery Temporal XML Query Language Copyright © 2003 Dengfeng Gao and Richard T. Snodgrass. All rights reserved.
Author(s)	Dengfeng Gao and Richard T. Snodgrass
Publication History	March 2003. A TIMECENTER Technical Report.

Contents

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 prove that τXQuery is XQuery-complete and sequenced queries are snapshot reducible to 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. Some example τXQuery queries are mapped to XQuery and are run on an XQuery engine, Galax. The query results obtained from different representations of the temporal information are snapshot equivalent.

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 [W3C02] 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 [ORA02], IBM [IBM02], and Microsoft [MS02], 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* [SA86]. While there has been some work addressing the transaction time dimension of XML data [CAM02, CTZ00, CTZ01, CTZ+02, MAC+01, ZCT01], 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 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. Two useful properties of τXQuery are identified in Section 6. The example τXQuery queries are evaluated on an XQuery engine, Galax, and their results are shown in Section 7. 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 9 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 10. Section 11 concludes the paper and lists interesting issues that are worthy of further study. Appendices A, B, and C provide the schema definition and the auxiliary functions used by the stratum respectively. Appendices D, E, F, G, and H give the non-temporal schema, the temporal annotation, the physical annotation, the representation schema, and the instance document of the example discussed in Section 2; these XML documents are used in the example in Section 7. Appendix I shows the temporal schema for the example generated by the stratum.

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 [AP02, GPT03]. 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” [Ahlert00], 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 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 periods of validity.

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 [W3C01], 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 doc-

```
<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

ument, 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., `customer`) must contain the period(s) of descendant elements (e.g., `action`). Note, though, that there is no such requirement between siblings, such as different `supportLevel`s 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.

```
<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="2002-9-19"/>
     ...
     <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 time-varying CRM XML document

The temporal XSchema model, τXSchema [CCD02] allows users to annotate XML Schemas to support temporal information while preserving data independence. The data designer starts by specifying the base non-temporal schema in XML schema. Then, he annotates the non-temporal schema to produce the *logical schema*, also termed the temporal annotated schema. These annotations state which components in the XML document can change over time. The remaining components of the XML document are considered to be static, and have the same values during the lifetime.

The designer must then further annotate the logical schema to create a *physical annotated schema* that states where in the time-varying document the timestamps should be placed, and how are the timestamps represented, which are 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.

The three schemas imply a representational schema that has the actual timestamps in all the right places. 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. All the four schemata and the instance temporal XML documents for the CRM example are given in Appendix D to H. We provide two physical schemata for the same logical schema, therefore, two representational schemata and two instance documents follow.

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 9.

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 non-temporal 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.

4.4 Other Kinds of Queries

Our approach, reminiscent of Böhlen’s *statement modifiers* for SQL [BJS00], can be extended to support other kinds of queries. Toman’s *point queries* [Tom98] provides yet another semantics; this could be effected with another modifier, `point`. The specific semantics of this and other possible modifiers is beyond the scope of this paper.

4.5 Compatibility

Representational queries have the same semantics (and syntax!) as XQuery. This indicates τXQuery is a superset of XQuery, which ensures τXQuery is *upward compatibility* with XQuery. Thus, any existing XQuery program can be run on a τXQuery processor without any changes.

If an existing application needs temporal support, the non-temporal schema can be augmented to include the `timestamp` and `timeVaryingAttribute` elements. *Temporal upward compatibility* [BBJS97] demands that existing XQuery code operating on the previous non-temporal documents continue to work (accessing the current state) on time-varying documents, without any changes.

So that τXQuery is both upward compatible and temporal upward compatible, we define two default working modes of the τXQuery processor, representational mode and current mode. When the default mode is set to representational, queries without any τXQuery reserved word are treated as representational queries. This ensures the upward compatibility. The default mode can be reset to current mode, in which queries without any τXQuery reserved word are treated as current queries. Then, representational queries need a reserved word (`representational validtime` or `rep validtime` for short) in current mode. The current mode ensures temporal upward compatibility.

The default mode can be configured by the user or it can be decided by the τXQuery processor automatically. Here is one possible way to determine the mode. The query processor accesses the documents that specified in the query to check if the namespace of the representational schema (`http://www.cs.arizona.edu/tau/RXSchema`) is used in any document. If that namespace is found, the document contains temporal data, and the default mode is set to current. Otherwise, the default mode is set to representational. Such an approach realizes both kinds of upward compatibility simultaneously.

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 [Stoy79].

The resulting XQuery expression uses some auxiliary functions; the definition of these functions is given in Appendix C. 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 9. 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 [W3C02] 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.

$\langle\text{TQ}\rangle ::= (\texttt{current} \mid \texttt{validtime}\ (\texttt{[}\langle\text{BT}\rangle\texttt{,}\ \langle\text{ET}\rangle\texttt{]})^{?} \mid \texttt{rep validtime}\ )^{?}\ \langle\text{Q}\rangle$

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. As mentioned in Section 4.5, τXQuery has two modes: the representational mode and the current mode, which supports the upward compatibility and the temporal upward compatibility respectively. Thus, we have two semantic functions ($rep[\![\ ]\!]$ and $cur[\![\ ]\!]$) to denote the semantics of τXQuery.

$\tau XQuery[\![\langle\text{TQ}\rangle]\!] = rep[\![\langle\text{TQ}\rangle]\!]$ or $cur[\![\langle\text{TQ}\rangle]\!]$

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.

$cur[\![\texttt{current}\ \langle\text{Q}\rangle]\!]\ =\ rep[\![\texttt{current}\ \langle\text{Q}\rangle]\!] = cur[\![\langle\text{Q}\rangle]\!]$

$\langle\text{Q}\rangle ::= \langle\text{QueryProlog}\rangle\ \langle\text{QueryBody}\rangle$

$cur[\![\langle\text{Q}\rangle]\!]\ =$

```
 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 $\langle$QueryProlog$\rangle$ (that is, the user-defined functions) and $\langle$QueryBody$\rangle$, 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()`.

$snapshot[\![\texttt{document(}\langle\text{String}\rangle\texttt{)}]\!]\ t = \texttt{tau:snapshot(document(}\langle\text{String}\rangle\texttt{),}\ t\texttt{)}$

$snapshot[\![\texttt{input()}]\!]\ t = \texttt{tau:snapshot(input(),}\ t\texttt{)}$

The auxiliary function `snapshot()` (see Appendix C) takes a node n and a time t as the input parameters and returns the snapshot of n at time t. This snapshot document has no valid timestamps; elements not valid now have been stripped out.

5.2 Representational Queries

The mapping for representational queries is trivial.

$rep[\![$`rep validtime` $\langle Q\rangle]\!] = rep[\![\langle Q\rangle]\!]$
$cur[\![$`rep validtime` $\langle Q\rangle]\!] = rep[\![\langle Q\rangle]\!]$
$rep[\![\langle Q\rangle]\!] = \langle Q\rangle$

The only thing needed is to remove the reserved word. 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. Sequenced queries have the same semantics in both current mode and representational mode.

$rep[\![$`validtime` $\langle Q\rangle]\!] = cur[\![$`validtime` $\langle Q\rangle]\!] =$
  $seq[\![\langle Q\rangle]\!]$ `$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. Therefore, this period is implementation dependent. The above semantic function is written with the assumption that the earliest time and the latest time can be represented by the system are `1000-01-01` and `9999-12-31` respectively.

If the valid-time period is explicitly specified by the user, the translation is as follows.

$rep[\![$`validtime [`$\langle BT\rangle$`,`$\langle ET\rangle$`]` $\langle Q\rangle]\!] = cur[\![$`validtime [`$\langle BT\rangle$`,`$\langle ET\rangle$`]` $\langle Q\rangle]\!] =$
  $seq[\![\langle Q\rangle]\!]$ `tau:period(`$\langle BT\rangle$`, `$\langle ET\rangle$`)`

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.

5.3.1 Semantics

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 straight-forward approach to map a sequenced query to XQuery, based on the following simple observation first made when the semantics of temporal aggregates were defined [SGM93]: 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. The semantic function of sequenced queries is as follows.

$\langle Q\rangle ::= \langle QueryProlog\rangle\ \langle QueryBody\rangle$
$seq[\![\langle Q\rangle]\!]\ p =$
```
 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"
```

$seq\,[\![\langle\text{QueryProlog}\rangle]\!]\,p$
```
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 in Appendix B. $getdoc\,[\![\;]\!]$ 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 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 in Appendix C, 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 9, we examine more sophisticated slicing strategies). Since the documents appearing in both ⟨QueryProlog⟩ and ⟨QueryBody⟩ need to be time-sliced, we define $seq\,[\![\langle\text{QueryProlog}\rangle]\!]\,p$ and $timeslice\,[\![\langle\text{QueryBody}\rangle]\!]\,t$ 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⟩}

$seq\,[\![\langle\text{FunctionDefn}\rangle]\!]\,p =$
`define function`⟨FuncName⟩`(xs:dateTime $tau:time,`⟨ParamList⟩$^?$`) returns` ⟨SequenceType⟩
{$timeslice\,[\![\langle\text{ExprSequence}\rangle]\!]$ `$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.

$timeslice\,[\![\langle\text{FunctionCall}\rangle]\!]\,t =$
⟨QName⟩`(`t`,` ($timeslice\,[\![\langle\text{Expr}\rangle]\!]\,t$ (`,` $timeslice\,[\![\langle\text{Expr}\rangle]\!]\,t$)*)$^?$`)`

The function `document()` and `input()` are the only two built-in functions that need to be mapped.

$timeslice\,[\![$`document(`⟨String⟩`)`$]\!]\,t =$ `tau:snapshot(document(`⟨String⟩`),` t`)`

Note that the actual parameter of `document()` could be an expression that evaluated to a string. In this case, the mapping approach does not work. However, we will give the mapping approach that can handle this case in Section 9.2.

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

$timeslice$ ⟦`input()`⟧ $t =$ `tau:snapshot(input(),` t`)`

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 need for an extra parameter for user-defined functions can be seen from an example. Let `term.xml` list (time-varying) terminology. A user-defined function `lookup` searches a term in this document and returns the definition.

```
define function lookup(xs:string $s) returns xs:node
{  document("term.xml")//term[name = $s]  }
```

During the mapping of function definitions, the constant periods are not known.

5.3.2 Typing Sequenced Queries

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()` as follows.

A single value of an atomic type: A single value with an atomic type is mapped to a sequence of elements with the type `tvv:timeVaryingValueType`.

An element whose value is a simple type: Such an element is mapped to a sequence of elements of the complex type with two subelements. One subelement is named `timestamp` with the type `rs:vtExtent`. The other is named `value` with the simple type of the original type of the element value.

An element of a complex type: This is mapped to a sequence of elements with a new complex type which extends the original complex type by adding a new subelement `timestamp`.

An attribute: An attribute is mapped to a sequence of elements, each of which is named `time-VaryingAttribute` and of `rs:vtAttributeTS` type.

A document: A document is mapped to a sequence of documents, the root element of which has one more subelement of the original root element, each with a `timestamp` subelement.

A processing-instruction, comment, or text node: These remain the same.

A sequence: A sequence is mapped to a sequence with each of its items mapped to the corresponding timestamped type.

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. We will discuss the order further in Section 8.

5.4 Summary

There are three modes in τXQuery. Representational queries are syntactically and semantically identical to XQuery queries. This is modulo the choice taken for the default mode. For the approach we advocate in Section 4.5, simultaneously ensuring upward compatibility and temporal upward compatibility requires that some XQuery expressions be interpreted in current mode. 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* [JD98, Snod87]. 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 Useful Properties of the Semantics

Based on the semantics defined in the last section, we identify the following two properties of τXQuery.

If every expression in a query language is equivalent to a valid XQuery expression, we term that query language *XQuery-complete*. For such languages, their syntactic constructs of τXQuery do not provide additional expressive power over XQuery.

Theorem 1 *τXQuery is XQuery-complete.*

Proof outline: We examine the semantics of the three kinds of queries. A representational query has the same semantics as the query without the keyword. A current query or sequenced query can be mapped to a semantically equivalent XQuery. This can be seen from the definitions of $\tau XQuery[\![\ ]\!]$, which produce valid XQuery strings except that the document name is a computed string. When the document name is a computed string, the mapping approach in Section 5.3 does not work. However, in Section 9.2, we will show two approaches that work in this case. Thus, we conclude τXQuery has the same expressive power as XQuery. □

To discuss the second property, we define the semantic function $eval[\![]\!]$ first. This function takes an XQuery string and a collection of XML documents (which we call the *input database*) as the inputs, and outputs the data resulting from the evaluation of the input query string against the input database.

We use c to denote a valid-time granule, and D to denote a temporal XML database. The snapshot function ss takes as arguments a valid-time temporal XML database D and a valid-time granule c and returns the snapshot of the XML document(s) that are valid at time c. τXQuery is *snapshot reducible to XQuery* if

$$\forall Q \in XQuery,\ \forall D,\ \forall c\ (ss(c, eval[\![seq[\![\texttt{validtime}\ \ Q]\!]]\!]\ D) = eval[\![Q]\!]\ ss(c, D)).$$

This is an application of snapshot reducibility defined on the relational algebra [Snod87].

Theorem 2 *τXQuery is snapshot reducible to XQuery.*

Proof outline: According to the semantic mapping defined in the last section, the sequenced query `validtime` Q is mapped to an XQuery expression evaluates Q on time-sliced portion of the input documents. The input documents are time-sliced on the constant periods. There are two cases regarding the relationship between c and the constant periods. In the first case, one of the constant periods contains c. Let cp denote this particular constant period. The snapshot of the result of the sequenced query at c is the result of Q executed on the input documents valid during cp. If cp = [bt, et], time-slicing a document on cp is done by taking snapshot of the document at bt, which yields the same document as the snapshot at c. Thus, the snapshot of the result of the sequenced query at c is the result of Q executed on the snapshot of the input documents at c. In the second case, none of the constant periods contains c. This implies that nothing in the input documents is valid at time c. Therefore, the result is an empty XML data set for both the left-hand-side and the right-hand-side of the above equality. □

7 Example Queries and Results

The three example queries mentioned in Section 4 have been mapped to XQuery and tested on Galax [Luc03]. Since Galax requires the syntax of the newest version of XQuery and it does not support all the feature of XQuery, we made a few changes to the auxiliary functions to enable the test. The changes falls in two categaries. One is the syntactic changes, e.g., the syntax of function definitions. The other changes are due to the incomplete implementation of Galax. For example, it does not support the data type `dataTime`. We just used `string` as a substitution.

As mentioned in Section 3, the physical representation of temporal XML data is independent from the logical schema of the same data. Different physical schemas imply different representational schemas, therefore, different structures of the temporal XML documents (instances). All the three queries are evaluated against the two different instances, `CRM1.xml` and `CRM2.xml` in Appendix H. They are defined by the two representational schemas in Appendix G, which are implied by the physical schemas in Appendix F respectively. The results of the queries on `CRM1.xml` are the same as those on `CRM2.xml`. Therefore, our mapping approach is independent from the physical schema of the temporal XML document. Indeed, except for some details of auxiliary XQuery functions defined in Appendix C, the formal semantics and the optimizations described later are largely independent of the representation.

7.1 Current Query

Query:
What is the average number of open support incidents per gold customer?
Expression:

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

Expanded Query:

```
import schema namespace rs = "http://www.cs.arizona.edu/tau/RXSchema"
       at "RXSchema.xsd"
declare namespace tau = "www.cs.arizona.edu/tau/Func"
define function tau:snapshot...
avg(for $c in tau:snapshot(document("CRM1.xml"), "2003-02-21")//customer
                                                     [@supportLevel="gold"]
    return count($c/supportIncident))
```

Result: `0`

The query results from `CRM1.xml` and `CRM2.xml` are the same.

7.2 Sequenced Query

The sequenced query is mapped using the slicing approach described in Section 5.3. The query results from the two instances are exactly the same. Both are not coalesced.

Query:
What is the history of the average number of open support incidents per gold customer?
Expression:

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

Expanded Query:

```
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"
define function tau:snapshot...
for $tau:p in tau:all-const-periods(tau:period("1000-01-01", "9999-12-31"),
                                    document("CRM1.xml")) return
  tau:associate-timestamp($tau:p,
                          avg(for $c in tau:snapshot(document("CRM1.xml"),
                              $tau:p/@vtBegin)//customer[@supportLevel="gold"]
                              return count($c/supportIncident)))
```

Result:

```
<timeVaryingValue>
  <rs:timestamp vtBegin="2001-01-05" vtEnd="2001-02-15"/>
  <value>0</value>
</timeVaryingValue>,
<timeVaryingValue>
  <rs:timestamp vtBegin="2001-02-15" vtEnd="2001-03-12"/>
  <value>0</value>
</timeVaryingValue>,
<timeVaryingValue>
  <rs:timestamp vtBegin="2001-03-12" vtEnd="2001-03-20"/>
  <value>0</value>
</timeVaryingValue>,
<timeVaryingValue>
  <rs:timestamp vtBegin="2001-03-20" vtEnd="2001-04-02"/>
  <value>0</value>
</timeVaryingValue>,
<timeVaryingValue>
  <rs:timestamp vtBegin="2001-04-02" vtEnd="2001-04-05"/>
  <value>1</value>
</timeVaryingValue>,
<timeVaryingValue>
  <rs:timestamp vtBegin="2001-04-05" vtEnd="2001-04-10"/>
  <value>0</value>
</timeVaryingValue>,
<timeVaryingValue>
  <rs:timestamp vtBegin="2001-04-10" vtEnd="2002-02-15"/>
  <value>0</value>
</timeVaryingValue>,
<timeVaryingValue>
  <rs:timestamp vtBegin="2002-02-15" vtEnd="2002-09-12"/>
  <value>0</value>
</timeVaryingValue>,
<timeVaryingValue>
  <rs:timestamp vtBegin="2002-09-12" vtEnd="2002-09-14"/>
  <value>1</value>
</timeVaryingValue>,
<timeVaryingValue>
  <rs:timestamp vtBegin="2002-09-14" vtEnd="forever"/>
  <value>0</value>
</timeVaryingValue>
```

7.3 Representational Query

The representational query must manipulate the temporal information explicitly. Therefore, different representational queries are written for different documents. The results of the two queries are the same.

Query:
What is the average number of support incidents, now or in the past, per gold customer, now or in the past?
Expression for CRM1.xml:

```
avg(for $c in document("CRM1.xml")//customer
    where $c/timeVaryingAttribute[@name="supportLevel][@value="gold"]
    return count($c/supportIncident))
```

Expression for CRM2.xml:

```
avg(for $n in distinct-values(document("CRM2.xml")//
                                customer[@supportLevel="gold"]//name)
return count(distinct-values(for $c in document("CRM2.xml")//customer
                             where $c/contactInfo/name=$n
                             return $c/supportIncident/product)))
```

Result: `1`

8 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, advocated by Torp [TJB97]. 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 and the current default mode of the stratum is consulted 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.

The two mappings are straight-forward. One interesting aspect is that all the semantic functions are implemented directly in the query mappers. For example, the $getdoc\llbracket\,\rrbracket$ semantic function discussed briefly in Section 5.3.1 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; 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 [BSS96, JD98]; 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.

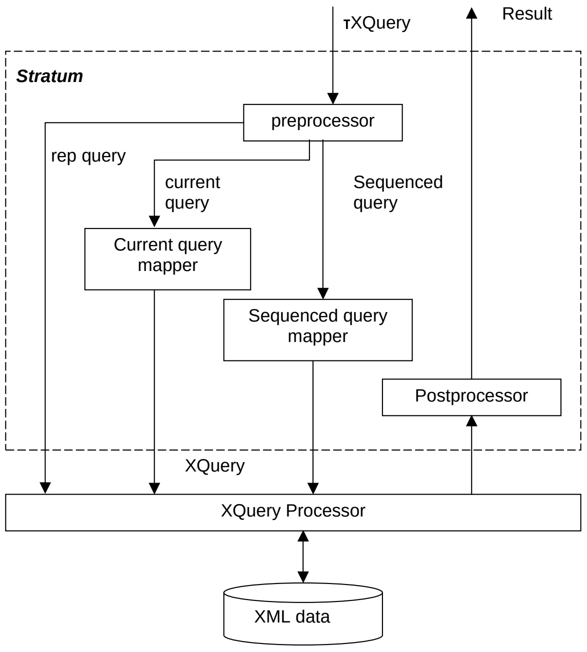


Figure 3: Architecture of the τXQuery stratum

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.

9 Optimization of Slicing

In Section 5.3, we presented one method to map sequenced τXQuery expressions to XQuery. In that 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 `timeVarying-Attribute` 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 [LM97]. Maximally-fragmenting still uses the periods in the data to compute the constant periods.)

Some queries 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.

9.1 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.

The semantic function $sn[\![\;]\!]\, p$ defines the mapping of sequenced queries.

$sn[\![\langle\mathrm{Q}\rangle]\!]\, p =$
```
 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"
 sn[[⟨QueryProlog⟩]] p
 define function tau:element-const-periods...
 ...
 for $tau:p in tau:element-const-periods(p, getdoc[[⟨Q⟩]], getnode[[⟨Q⟩]])
 return tau:associate-timestamp($tau:p, timeslice[[⟨QueryBody⟩]] $tau:p/@vtBegin)
```

The only difference between selected node slicing and maximally-fragmented slicing is that it uses `element-const-periods()` rather than `all-const-periods()`. The XQuery function `element-const-periods()` takes a sequence of documents and a sequence of strings representing node names (elements or attributes) and collects 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. The function $getnodes[\![\;]\!]$ implemented in the stratum takes a query string as the input and returns the node names that appear in the query string.

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 function `element-const-periods()` will not collect the valid periods of the element `action`.

Selected node slicing does not work when a wildcard is present in the query and the schema information is not available, However, a sample of four existing XML benchmarks shows that wildcards appear only in a small number of queries: none in XMark [XMark], one in 23 queries in XOO7 [XOO7], one in 20 queries in XBench [XBench], and four in eight queries in XMach-1 [XMach].

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.

9.2 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 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.

9.2.1 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. The translation rule for each production is given in this section. Since any XQuery program can be normalized by using the core grammar, a subset of the XQuery grammar provided by the W3C, defining the semantics to map the core grammar of τXQuery is sufficient.

Consider the example query ”what is the history of the average number of current 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 [W3C02]. The only difference is we change the prefix

```
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

`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.

From now on, we will show the BNF of core grammar and the mapping of each production in the core grammar. Normalization is performed 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. This description is somewhat involved, because the time-slicing is done individually for each nonterminal in the core grammar.

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

   As with the mapping function defined in previous sections, before $\langle$QueryProlog$\rangle$ and $\langle$QueryBody$\rangle$ are mapped, some necessary schema imports, namespace declarations, and function definitions that help the sequenced mapping should be put at the beginning. We have seen `rs:vtExtent` and `tvv:timeVaryingValueType` previously. The type `timeVaryingValueType` is the timestamped analogue of all the built-in simple types. This type is used for substitution of the original data types referenced in the query body (especially in `typeswitch` expression and the signature of functions). We will examine the details later. The `tau` namespace also contains the sequenced version of the built-in operations and functions such as `xf:avg()` and `op:numeric-add()`.

   $cb\,[\![\langle\text{Q}\rangle]\!]\,p =$
   ```
    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"
```
$cb\,[\![\langle\text{QueryProlog}\rangle]\!]\;p$
```
define function tau:snapshot...
...
```
$cb\,[\![\langle\text{QueryBody}\rangle]\!]\;p$

2. $\langle$QueryProlog$\rangle$ ::= ($\langle$NamespaceDecl$\rangle$
   | $\langle$XMLSpaceDecl$\rangle$
   | $\langle$DefaultNamespaceDecl$\rangle$
   | $\langle$DefaultCollationDecl$\rangle$
   | $\langle$SchemaImport$\rangle$)* $\langle$FunctionDefn$\rangle^*$

   $\langle$XMLSpaceDecl$\rangle$ and $\langle$DefaultCollationDecl$\rangle$ do not need mapping. All the rest require work. The namespace declaration produces an environment that associates a prefix with a URI, whose schema location is indicated by the schema import statement. The τXQuery processor maps the namespace declaration to its temporal counterpart. For example, the following statement declares a namespace `crm` defined by `CRM.xsd`.

   ```
   import schema namespace crm = "CRM"  at "CRM.xsd"
   ```

   This declaration will be translated to the following.

   ```
   import schema namespace tcrm =
   "http://www.cs.arizona.edu/stratum/tCRM"
          at "tCRM.xsd"
   ```

   The file `tCRM.xsd` is a new schema file generated from `CRM.xsd`, but with all the user-defined data type timestamped. We call it the *timestamp schema*. This schema is similar to the schema that defines `tvv:timeVaryingValueType`. The timestamp schema of the CRM example is given in Appendix I. The data types defined in `tCRM.xsd` will be used when the sequenced query specifies data types defined in `CRM.xsd`. The namespace `tcrm` replaces the namespace `crm` in the sequenced query.

   As an example, consider the type of `customer` in `CRM.xml` defined as `crm:customerType`. The timestamp schema `tCRM.xsd` defines another type `tcrm:customerType`. An element of this new type have all the attributes and subelements of `crm:customerType`, along with zero or more `timestamp` and `timeVaryingAttribute` which could appear as children of `customer` and all its subelements.

   The $\langle$DefaultNamespaceDecl$\rangle$ is processed similarly. In this way, it is guaranteed the user-defined type can be validated correctly in the sequenced semantics. Consider the following statement that declares a default namespace `crm` defined by `CRM.xsd`.

   ```
   import schema default element namespace
          crm ="CRM" at "CRM.xsd"
   ```

   The declaration is translated to the following.

   ```
   import schema default element namespace
          tcrm ="http://www.cs.arizona.edu/stratum/tCRM"
          at "tCRM.xsd"
   ```

   When the namespace `crm` is specified in the query, it is replaced with `tcrm`. If no namespace prefix is specified for an element in the query, the query processor considers the element in the default namespace.

3. $\langle$FunctionDefn$\rangle$ ::= `define function` $\langle$FuncName$\rangle$ `(`$\langle$ParamList$\rangle^?$`)`
`returns` $\langle$SequenceType$\rangle$
`{`$\langle$ExprSequence$\rangle$`}`

$cb\,[\![\langle\text{FunctionDefn}\rangle]\!]\;p =$
`define function` $\langle$FuncName$\rangle$ `(` $(\,cb\,[\![\langle\text{ParamList}\rangle]\!]\;p\,)^?$ `)`
`returns` $cb\,[\![\langle\text{SequenceType}\rangle]\!]\;p$
$\{cb\,[\![\langle\text{ExprSequence}\rangle]\!]\;p\}$

$\langle$ParamList$\rangle$ ::= $\langle$Param$\rangle$ (`,` $\langle$Param$\rangle$)*
$cb\,[\![\langle\text{ParamList}\rangle]\!]\;p = cb\,[\![\langle\text{Param}\rangle]\!]\;p$ (`,` $cb\,[\![\langle\text{Param}\rangle]\!]\;p\,)^*$
For the rest of the paper, we will omit such obvious semantic functions that mirror the productions.

$\langle$Param$\rangle$ ::= $\langle$SequenceType$\rangle$ `$`$\langle$VarName$\rangle$
$cb\,[\![\langle\text{Param}\rangle]\!]\;p = cb\,[\![\langle\text{SequenceType}\rangle]\!]\;p$ `$`$\langle$VarName$\rangle$

A function defined by a user should be evaluated using sequenced semantics. However, the data type of the input parameters and the return value may be the data types without timestamps, since the user may not have annotated the particular data type. If the non-temporal signature of the function is retained, the valid time period of the input expression will be lost. Thus, in such cases the result of the function call doesn't comply with the sequenced semantics of the function. Changing the signature of the function by replacing the non-temporal types with temporal types defined in the timestamp schema will solve this problem.

$\langle$SequenceType$\rangle$ ::= ($\langle$ItemType$\rangle\langle$OccurenceIndicator$\rangle$) | `empty`

$cb\,[\![\langle\text{ItemType}\rangle\;\langle\text{OccurenceIndicator}\rangle]\!]\;p = temType\,[\![\langle\text{ItemType}\rangle]\!]\;\;\langle\text{OccurenceIndicator}\rangle$

The new semantic function $temType\,[\![\;]\!]$ takes a string representing a non-temporal type as an input parameter and returns a string representing the corresponding timestamped type. Note that the period p is not passed to this function because this mapping does not depend on a particular period.

$\langle$ItemType$\rangle$ ::= ((`element` | `attribute`)$\langle$ElemOrAttrType$\rangle^?$)
| $\langle$AtomicType$\rangle$
| `node`
| `processing-instruction`
| `comment`
| `text`
| `document`
| `item`
| `untyped`
| `atomic value`

$\langle$AtomicType$\rangle$, `atomic value`, and `untyped` are mapped to `tvv:timeVaryingValueType`. An `element` with the type specified is converted to its timestamped counterpart. For example, `element of type crm:customerType` is converted to `element of type tcrm:customerType`. An `attribute` is always mapped to an `element` of the type `rs:vtAttributeTS`. The remaining data types retain their XQuery semantics.

4. $\langle$QueryBody$\rangle$ ::= $\langle$ExprSequence$\rangle^?$
$\langle$ExprSequence$\rangle$ ::= $\langle$Expr$\rangle$ (`,`$\langle$Expr$\rangle$)*
$\langle$Expr$\rangle$ ::= $\langle$UnorderedExpr$\rangle$ (`stable`$^?$ `sortby(`$\langle$SortSpecList$\rangle$`)`)*
The semantics of sort in XQuery has not yet been decided, and is not defined formally in the working draft [W3C02]. We discussed ordering briefly in Section 5.3.2. We leave it for future work.

5. ⟨UnorderedExpr⟩ ::= `unordered`$^?$ (⟨ForExpr⟩ | ⟨LetExpr⟩)
   ⟨ForExpr⟩ ::= (⟨ForClause⟩ `return`)* ⟨TypeswitchExpr⟩
   ⟨ForClause⟩ ::= `for` ⟨SequenceType⟩$^?$ `$`⟨VarName⟩ `in` ⟨Expr⟩

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

   The auxiliary function `periods-of()` returns all the timestamps associated with the input node. Since the sequence returned by ⟨Expr⟩ could contain multiple versions of an item, valid over different periods of time, the following ⟨TypeswitchExpr⟩ should be evaluated in each of these periods. The function `copy-restricted-subtree()` makes a copy of the input node (nodes) and removes the subtrees that are not valid in the input period.

6. ⟨LetExpr⟩ ::= (⟨LetClause⟩ `return`)* ⟨TypeswitchExpr⟩
   ⟨LetClause⟩ ::= `let` ⟨SequenceType⟩$^?$ `$`⟨VarName⟩ `:=` ⟨Expr⟩

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

   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 auxiliary 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. Thus, the periods returned in this level could be divided further into smaller constant periods.

7. ⟨TypeswitchExpr⟩ ::= (`typeswitch (`⟨Expr⟩`)`
                          (`case` ⟨SequenceType⟩ `$`⟨VarName⟩ `return` ⟨Expr$_1$⟩)$^+$
                           `default $`⟨VarName⟩ `return`)* ⟨IfExpr⟩

```
cb⟦⟨TypeswitchExpr⟩⟧ p =
  let $tau:s := cb⟦⟨Expr⟩⟧ p
  for $tau:p in tau:const-periods(p, $tau:s)
  let $tau:v := tau:copy-restricted-subtree($tau:p, $tau:s)
  return
    typeswitch ($tau:v)
      (case cb⟦⟨SequenceType⟩⟧ $tau:p $⟨VarName⟩ return cb⟦⟨Expr1⟩⟧ $tau:p )+
      default $⟨VarName⟩ return cb⟦⟨IfExpr⟩⟧ $tau:p
```

8. ⟨IfExpr⟩ ::= (`if (`⟨Expr$_1$⟩`) then` ⟨Expr$_2$⟩ `else`)* ⟨ValueExpr⟩

```
cb⟦⟨IfExpr⟩⟧ p =
  let $tau:b := cb⟦⟨Expr1⟩⟧ p
  for $tau:p in tau:const-periods(p, $tau:b)
  let $tau:s := snapshot(tau:copy-restricted-subtree($tau:p, $tau:b),
                         $tau:p/@vtBegin) return
```

```
    if ($tau:s)
    then $cb\,[\![\langle\text{Expr}_2\rangle]\!]$ $tau:p
    else $cb\,[\![\langle\text{ValueExpr}\rangle]\!]$ $tau:p
```

In XQuery, ⟨IfExpr⟩ evaluates an expression to a Boolean value and chooses the branch according to that value. The rule to evaluate the expression to a Boolean value is complicated by the fact that the result of the expression may be time-varying. This is why we time-slice the expression to be evaluated to a Boolean value and then evaluate the snapshot of the expression over each of the constant periods.

9. ⟨ValueExpr⟩ ::= ⟨ValidateExpr⟩ | ⟨CastExpr⟩ | ⟨Constructor⟩ | ⟨PathExpr⟩
⟨ValidateExpr⟩ ::= `validate` ⟨SchemaContext⟩$^?$ { ⟨Expr⟩ }

```
$cb\,[\![\langle\text{ValidateExpr}\rangle]\!]\; p =$
  let $tau:s := $cb\,[\![\langle\text{Expr}\rangle]\!]\; p$
  for $tau:p in tau:const-periods($p$, $tau:s) return
    validate ( $temSC\,[\![\langle\text{SchemaContext}\rangle]\!]$ )$^?$ tau:copy-restricted-subtree(
                                              $tau:p, $tau:s)
```

⟨SchemaContext⟩ = `in` ⟨SchemaGlobalContext⟩ (`/`⟨SchemaContextStep⟩)*
⟨SchemaGlobalContext⟩ = ⟨QName⟩ | `type` ⟨QName⟩
⟨SchemaContextStep⟩ = ⟨QName⟩

```
$temSC\,[\![\langle\text{SchemaContext}\rangle]\!]$ =
  in $temSC\,[\![\langle\text{SchemaGlobalContext}\rangle]\!]$ ( /$temSC\,[\![\langle\text{SchemaContextStep}\rangle]\!]$ )$^*$
```

The new function $temSC\,[\![\;]\!]$ maps a string which is a name of an element, attribute, or type, to its timestamped anolog. This function is similar to $temType\,[\![\;]\!]$. The timestamp of the node will not be lost after it is validated since the non-temporal schema context is replaced by the corresponding temporal schema context. An example of ⟨ValidateExpr⟩ is as follows (suppose that `$x` is bound to a `product` element).

```
validate in crm:customer/supportIncident $x
```

The ⟨SchemaContext⟩ portion is mapped to the following.

```
tcrm:customer/supportIncident
```

10. ⟨CastExpr⟩ ::= `cast as` ⟨SequenceType⟩ (⟨ExprSequence⟩$^?$)

```
$cb\,[\![\langle\text{CastExpr}\rangle]\!]\; p =$
  let $tau:s := $cb\,[\![\langle\text{ExprSequence}\rangle]\!]\; p$
  for $tau:p in tau:const-periods($p$, $tau:s)
  let $tau:v := cast as ⟨SequenceType⟩ snapshot(
     tau:copy-restricted-subtree($tau:p, $tau:s), $tau:p/@vtBegin)
  where not empty($tau:v)
  return <timeVaryingValue>
             $tau:p
             <value>$tau:v</value>
         </timeVaryingValue>
```

Since the ⟨CastExpr⟩ can only cast an expression of one simple type to another simple type, we cast the snapshot of the expression at each constant period and wrap the cast result in a `timeVaryingValue` element.

11. ⟨Constructor⟩ ::= ⟨XmlComment⟩
    | ⟨XmlProcessingInstruction⟩
    | ⟨ComputedDocumentConstructor⟩
    | ⟨ComputedElementConstructor⟩
    | ⟨ComputedAttributeConstructor⟩

    Only computed constructors have a sequenced semantics different from their XQuery semantics.

```
⟨ComputedDocumentConstructor⟩ ::=
  document { ⟨ExprSequence⟩ }
```

```
cb ⟦⟨ComputedDocumentConstructor⟩⟧ p =
  document
  { element timeVaryingRoot
      {  p,  cb ⟦⟨ExprSequence⟩⟧ p    }
  }
```

One `timeVaryingRoot` element is added to each computed document as the root element. This is again because the expression sequence is time-varying. Without `timeVaryingRoot`, multiple versions of the root will violate the well-formedness of a document.

```
⟨ComputedElementConstructor⟩ ::= element ⟨QName⟩ { ⟨ExprSequence⟩? }
                               | element { ⟨Expr⟩}  {⟨ExprSequence⟩?}
cb ⟦element ⟨QName⟩ {⟨ExprSequence⟩}⟧ p =
  element ⟨QName⟩ {  p,  cb ⟦⟨ExprSequence⟩⟧ p  }
```

The mapping of a computed element constructor adds a timestamp to the element and evaluates the expression sequence using the sequenced semantics.

```
cb ⟦element ⟨Expr⟩ {⟨ExprSequence⟩}⟧ p =
  let $tau:s := cb ⟦⟨Expr⟩⟧ p
  for $tau:p in $tau:const-periods(p, $tau:s)
  return
    element {snapshot(tau:copy-restricted-subtree($tau:p, $tau:s),
                         $tau:p/@vtBegin)}
    {
      $tau:p,
      cb ⟦⟨ExprSequence⟩⟧ $tau:p
    }
```

```
⟨ComputedAttributeConstructor⟩ ::= attribute ⟨QName⟩ {⟨ExprSequence⟩?}
                                 | attribute {⟨Expr⟩} {⟨ExprSequence⟩?}
cb ⟦⟨ComputedAttributeConstructor⟩⟧ p =
  let $tau:s := cb ⟦⟨ExprSequence⟩⟧ p
  for $tau:p in $tau:const-periods(p, $tau:s) return
    element timeVaryingAttribute
    {
      attribute name {⟨QName⟩},
      attribute value {snapshot(tau:copy-restricted-subtree($tau:p,
                                   $tau:s), $tau:p/@vtBegin)},
      attribute vtBegin {$tau:p/@vtBegin},
      attribute vtEnd {$tau:p/@vtEnd}
    }
```

An attribute constructor is mapped to construct a `timeVaryingAttribute` element.

12. ⟨PathExpr⟩ ::= ⟨StepExpr⟩
⟨StepExpr⟩ ::= `$`⟨VarName⟩`/`⟨ForwardStep⟩ | `$`⟨VarName⟩`/`⟨ReverseStep⟩ | ⟨PrimaryExpr⟩
⟨ForwardStep⟩ ::= ⟨ForwardAxis⟩ ⟨NodeTest⟩

```
⟨ForwardAxis⟩ ::= child ::
                 |descendant ::
                 |attribute ::
                 |self ::
                 |descendant-or-self ::
                 |following-sibling ::
                 |following ::
                 |namespace ::
```

⟨NodeTest⟩ ::= ⟨KindTest⟩ | ⟨NameTest⟩

```
⟨KindTest⟩ ::= processing-instruction(⟨StringLiteral⟩?)
              |comment()
              |text()
              |node()
```

⟨KindTest⟩ is mapped to itself.

⟨NameTest⟩ ::= ⟨QName⟩ | ⟨Wildcard⟩

Among the forward axes, the attribute axis is special because all the attributes are mapped to elements. The following function gives the mapping rule for attributes.

```
cb ⟦$⟨VarName⟩/attribute::⟨NameTest⟩⟧ p =
 (for $tau:a in $⟨VarName⟩/@⟨NameTest⟩
  return
    element timeVaryingAttribute
    {
      attribute name {name($tau:a)},
      attribute value {data($tau:a)},
      attribute vtBegin {p/@vtBegin},
      attribute vtEnd {p/@vtEnd}
    },
  for $tau:ta in $⟨VarName⟩/timeVaryingAttribute[@name = ⟨NameTest⟩]
  return tau:copy-restricted-subtree(p, $tau:ta)
```

The predicate in the second `for` expression ensures that the the valid period of the time-varying attribute overlaps the input period. The function `copy-restricted-subtree()` guarantees the valid period of the returned time-varying attribute is the intersection of the period of the original time-varying attribute and that of the input period.

The other forward steps are mapped as follows. The filters in the where clause hide all the subelements added by τXQuery from the user.

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

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. Only when the ⟨NodeTest⟩

is a ⟨NameTest⟩ and it has the format of ⟨Prefix⟩:⟨LocalName⟩, does it need to be mapped to sequenced semantics. The new function $temNode⟦\,⟧$ takes the string representing the name of a namespace and returns the corresponding temporal namespace.

cb ⟦⟨Prefix⟩:⟨LocalName⟩⟧ p = $temNode$ ⟦⟨Prefix⟩⟧ : ⟨LocalName⟩

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 and sibling axis in path expression of the original node. In the next section, we will introduce a per-expression slicing approach that can handle all the path expressions.

⟨ReverseStep⟩ ::= ⟨ReverseAxis⟩⟨NodeTest⟩

```
⟨ReverseAxis⟩ ::= parent::
               |ancestor::
               |preceding-sibling::
               |preceding::
               |ancestor-or-self::
```

13. ⟨PrimaryExpr⟩ ::= ⟨Literal⟩
    | ⟨FunctionCall⟩
    | $⟨VarName⟩
    | (⟨ExprSequence⟩?)

cb ⟦⟨Literal⟩⟧ p =
```
  <timeVaryingValue>
    p,
    <value>⟨Literal⟩</value>
  </timeVaryingValue>
```

A ⟨Literal⟩ is mapped to a `timeVaryingValue` element.

14. ⟨FunctionCall⟩ ::= ⟨QName⟩ ((⟨Expr$_1$⟩ (, ⟨Expr$_2$⟩)*)?)

Functions in τXQuery are divided into two groups. Each group of functions are treated differently from others when they are called.

The first group are user-defined functions, which are mapped as follows.

cb ⟦⟨FunctionCall⟩⟧ p = ⟨QName⟩((cb ⟦⟨Expr$_1$⟩⟧ p(, cb ⟦⟨Expr$_2$⟩⟧ p)*)?)

The second group are built-in functions. These function calls can be mapped 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.

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)))
```

Some built-in functions, cannot be given a sequenced semantics because the identity information is lost when the nodes are copied during the evaluation. These functions are listed below.

```
xf:base-uri
xf:lang
xf:root
xf:id
xf:idref
op:node-equal
xf:distinct-nodes
```

15. $cb\,[\![\$\langle\text{VarName}\rangle]\!]\; p =$
    `tau:copy-restricted-subtree(`p`, $`⟨VarName⟩`)`

    The function `copy-restricted-subtree()` takes one or more time periods and a variable as input parameters. It propagates the time period from the top node of the variable to all its descendants, while removing elements not valid during the input periods.

Given the example query stated at the beginning of this section and repeated in Figure 5, the τXQuery processor first normalizes it to the query shown in Figure 4. This normalized query is then mapped to the XQuery query in Figure 5. The document trees (or sub-trees) are time-sliced at each level of the expression on the constant periods of the root of the trees (or sub-trees). A copy of the intermediate result is made on each constant period by `copy-restricted-subtree()`. When the evaluation goes to a deeper level of the expression, the intermediate result is time-sliced further either because the evaluation period changes or because the context nodes are in a deeper level of the document trees.

9.2.2 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.

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 6. Suppose, a sub-tree rooted at A without timestamps has two timestamped sub-elements B and C (Figure 6(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 6(b), while in-place slicing returns the original sub-tree with an actual timestamp as shown in Figure 6(c).

As in the last section, we will show the translation for in-place slicing production by production. The semantic function that defines the mapping is called $inp\,[\![\;]\!]\,p$. It is helpful to compare each production with the analogous definition of $cb\,[\![\;]\!]$.

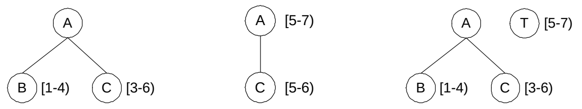
1. $\langle\text{Q}\rangle ::= \langle\text{QueryProlog}\rangle\ \langle\text{QueryBody}\rangle$

   $inp\,[\![\langle\text{Q}\rangle]\!]\; p =$
   ```
   import schema namespace
          rs = "http://www.cs.arizona.edu/tau/RXSchema"
          at "RXSchema.xsd"
   declare namespace tau = "www.cs.arizona.edu/tau/Func"
   ```
   $inp\,[\![\langle\text{QueryProlog}\rangle]\!]\; p$
   ```
   define function tau:apply-timestamp...
   ...
   ```
   `tau:apply-timestamp(`$inp\,[\![\langle\text{QueryBody}\rangle]\!]\; p$`)`

   Since the result of $inp\,[\![\langle\text{QueryBody}\rangle]\!]\; p$ is a sequence of items and their timestamps, One more step over $cb\,[\![\;]\!]$ is needed to get the desired result. The function `apply-timestamp()` makes a copy of the final result with the correct timestamps.

```
{-- the original tau XQuery:
    validtime avg(for $c in document("CRM.xml")//customer return
                      count($c/supportIncident))
    the normalized query is:
    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))) --}
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"
define function tau:const-periods...
...
{-- validtime avg(for $c in --}
let $tau:par :=
  (for $tau:i in
   {-- let $tau:sequence:=document("CRM.xml") return --}
    (let $tau:s := (let $tau:par1 := element timeVaryingValue
                                        tau:period("1000-01-01", "9999-12-31"),
                                         <value>CRM.xml</value>
                    for $tau:p in tau:all-const-periods(tau:period("1000-01-01",
                                             "9999-12-31"), $tau:par1) return
                      tau:associate-timestamp($tau:p,
                                 document(tau:snapshot($tau:par1, $tau:p/@vtBegin)))
     for $tau:p in tau:const-periods(tau:period("1000-01-01", "9999-12-31"), $tau:s)
     let $tau:sequence := tau:copy-restricted-subtree($tau:p, $tau:s) return
       {-- for $tau:dot in $tau:sequence return --}
       for $tau:i1 in tau:copy-restricted-subtree($tau:p, $tau:sequence)
       for $tau:p1 in tau:periods-of($tau:i1)
       let $tau:dot := tau:copy-restricted-subtree($tau:p1, $tau:i1) return
         {-- $tau:dot/descendant-or-self::customer --}
         for $tau:step in $tau:dot/descendant-or-self::customer
         where not(tau:special-node($tau:step)) return
           tau:copy-restricted-subtree($tau:p1, $tau:step))
   for $tau:p in tau:periods-of($tau:i)
   let $c := tau:copy-restricted-subtree($tau:p, $tau:i) return
     {-- count(let $tau:sequence := $c return --}
     let $tau:par1 :=
       (let $tau:s1 := tau:copy-restricted-subtree($tau:p, $c)
        for $tau:p1 in tau:const-periods($tau:p, $tau:s1)
        let $tau:sequence := tau:copy-restricted-subtree($tau:p1, $tau:s1) return
          {-- for $tau:dot in $tau:sequence return --}
          for $tau:i1 in tau:copy-restricted-subtree($tau:p1, $tau:sequence)
          for $tau:p2 in tau:periods-of($tau:i1)
          let $tau:dot := tau:copy-restricted-subtree($tau:p2, $tau:i1) return
            {-- $tau:dot/child::supportIncident))) --}
            for $tau:step in $tau:dot/child::supportIncident
            where not(tau:special-node($tau:step)) return
              tau:copy-restricted-subtree($tau:p2, $tau:step))
     for $tau:p3 in tau:all-const-periods($tau:p, $tau:par1) return
       tau:associate-timestamp($tau:p,
                                count(tau:snapshot($tau:par1, $tau:p/@vtBegin))))
for $tau:p in tau:all-const-periods(tau:period("1000-01-01", "9999-12-31"), $tau:par)
return tau:associate-timestamp($tau:p, avg(tau:snapshot($tau:par, $tau:p/@vtBegin)))
```

Figure 5: The Result of Copy-Based Per-Expression Slicing



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

Figure 6: Intermediate results for per-expression slicing

2. $\langle$QueryProlog$\rangle$ ::= ($\langle$NamespaceDecl$\rangle$
   | $\langle$XMLSpaceDecl$\rangle$
   | $\langle$DefaultNamespaceDecl$\rangle$
   | $\langle$DefaultCollationDecl$\rangle$
   | $\langle$SchemaImport$\rangle$)* $\langle$FunctionDefn$\rangle$*

   Among the non-terminals on the right-hand-side, only $\langle$FunctionDefn$\rangle$ need to be translated.

   $\langle$FunctionDefn$\rangle$ ::= `define function` $\langle$FuncName$\rangle$ `(`$\langle$ParamList$\rangle^?$`)`
   `returns` $\langle$SequenceType$\rangle$
   `{`$\langle$ExprSequence$\rangle$`}`

   ```
   inp [[⟨FunctionDefn⟩]] p =
     define function ⟨FuncName⟩ ( ( inp [[⟨ParamList⟩]] p )? )  returns item*
     {inp [[⟨ExprSequence⟩]] p}
   ```

   The signature of each user-defined function is changed from *cb* ⟦ ⟧ so that all the input and output data types are `item*` no matter what type they are in the original query. The reason is the intermediate result is always a sequence of items and their timestamps.

3. $\langle$QueryBody$\rangle$ ::= $\langle$ExprSequence$\rangle^?$

   $\langle$ExprSequence$\rangle$ ::= $\langle$Expr$\rangle$ (`,`$\langle$Expr$\rangle$)*

   $\langle$Expr$\rangle$ ::= $\langle$UnorderedExpr$\rangle$ (`stable`$^?$ `sortby(`$\langle$SortSpecList$\rangle$`))`*

   ```
   inp [[⟨Expr⟩]] p =
     let $tau:s := inp [[⟨UnorderedExpr⟩]] p
     for $tau:i in (tau:get-actual-items($tau:s) sortby ⟨SortSpecList⟩)
     return ($tau:i, item-at($tau:s, index-of($tau:s, $tau:i) + 1)
   ```

   The `sortby` operation changes the ordering of the items in the intermediate results. The mapping function must make sure the timestamp is immediately after the corresponding item. The function `get-actual-items()` takes a sequence and returns only the items in the odd position.

4. $\langle$UnorderedExpr$\rangle$ ::= `unordered`$^?$ ($\langle$ForExpr$\rangle$ | $\langle$LetExpr$\rangle$)

   $\langle$ForExpr$\rangle$ ::= ($\langle$ForClause$\rangle$ `return`)* $\langle$TypeswitchExpr$\rangle$

   $\langle$ForClause$\rangle$ ::= `for` $\langle$SequenceType$\rangle^?$ `$`$\langle$VarName$\rangle$ `in` $\langle$Expr$\rangle$

   ```
   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
   ```

   The variable $\langle$VarName$\rangle$ is bound to an (item, timestamp) pair instead of a single item in *cb* ⟦ ⟧.

5. ⟨LetExpr⟩ ::= (⟨LetClause⟩ `return`)* ⟨TypeswitchExpr⟩
   ⟨LetClause⟩ ::= `let` ⟨SequenceType⟩$^?$ `$`⟨VarName⟩ `:=` ⟨Expr⟩

   inp ⟦⟨LetExpr⟩⟧ p =
```
  let $tau:s := inp⟦⟨Expr⟩⟧ p
  for $tau:p in tau:const-periods2(p, $tau:s)
  let $⟨VarName⟩ := tau:sequence-in-period($tau:s, $tau:p)
  return inp⟦⟨TypeswitchExpr⟩⟧ $tau:p
```
   The function `const-periods2()` takes a sequence, including items and their timestamps, and a period as inputs. It returns the constant periods of this sequence of items contained in the input period. The function `sequence-in-period()` takes two input parameters, a sequence of items with their timestamps and a period. It computes the overlap of the valid period of each item and the input period. Those items that are not valid in the input period are filtered out. The rest items with the overlapped periods are returned in a sequence.

6. ⟨TypeswitchExpr⟩ ::= (`typeswitch (`⟨Expr⟩`)`
   (`case` ⟨SequenceType⟩ `$`⟨VarName⟩ `return` ⟨Expr$_1$⟩)$^+$
   `default $`⟨VarName⟩ `return`)* ⟨IfExpr⟩

   seq ⟦⟨TypeswitchExpr⟩⟧ p =
```
  let $tau:s := inp⟦⟨Expr⟩⟧ p
  for $tau:p in tau:const-periods2(p, $tau:s)
  let $tau:ss := tau:sequence-in-period($tau:s, $tau:p)
  let $tau:ssp := tau:get-periods($tau:ss) return
    typeswitch (tau:get-actual-items($tau:ss))
      (case ⟨SequenceType⟩ $tau:v return
          let $⟨VarName⟩ := tau:interleave($tau:v, $tau:ssp) return
            inp⟦⟨Expr1⟩⟧ $tau:p)+
        default $tau:v return
          let $⟨VarName⟩ := tau:interleave($tau:v, $tau:ssp) return
            inp⟦⟨IfExpr⟩⟧ $tau:p
```
   When the type of the expression is examined, the actual items are extracted from the sequence. Before the result is returned, the timestamps and the actual items are interleaved in one sequence. The function `get-periods()` takes a sequence and returns the timestamps in the even position as a sequence. The function `interleave()` takes two sequences as inputs and interleaves them as one sequence.

7. ⟨IfExpr⟩ ::= (`if (`⟨Expr$_1$⟩`) then` ⟨Expr$_2$⟩ `else`)* ⟨ValueExpr⟩

   inp ⟦⟨IfExpr⟩⟧ p =
```
  let $tau:s := inp⟦⟨Expr1⟩⟧ p
  for $tau:p in tau:const-periods2(p, $tau:s) return
    if (tau:get-actual-items(tau:sequence-in-period($tau:s, $tau:p)))
    then inp⟦⟨Expr2⟩⟧ $tau:p
    else inp⟦⟨ValueExpr⟩⟧ $tau:p
```

8. ⟨ValueExpr⟩ ::= ⟨ValidateExpr⟩ | ⟨CastExpr⟩ | ⟨Constructor⟩ | ⟨PathExpr⟩
   ⟨ValidateExpr⟩ ::= `validate` ⟨SchemaContext⟩$^?$ { ⟨Expr⟩ }

   inp ⟦⟨ValidateExpr⟩⟧ p =
```
  let $tau:s := inp⟦⟨Expr⟩⟧ p
  for $tau:p in tau:const-periods2(p, $tau:s)
  let $tau:ss := tau:sequence-in-period($tau:s, $tau:p)
  let $tau:v := validate ⟨SchemaContext⟩? tau:get-actual-items($tau:ss)
  return tau:interleave($tau:v, get-periods($tau:ss))
```

9. ⟨CastExpr⟩ ::= `cast as` ⟨SequenceType⟩ `(`⟨ExprSequence⟩$^?$`)`

```
inp⟦⟨CastExpr⟩⟧ p =
  let $tau:s := inp⟦⟨Expr⟩⟧ p
  for $tau:p in tau:const-periods2(p, $tau:s)
  let $tau:ss := tau:sequence-in-period($tau:s, $tau:p)
  let $tau:v := cast as ⟨SequenceType⟩ tau:get-actual-items($tau:ss)
  return tau:interleave($tau:v, get-periods($tau:ss))
```

10. ⟨Constructor⟩ ::= ⟨XmlComment⟩
    | ⟨XmlProcessingInstruction⟩
    | ⟨ComputedDocumentConstructor⟩
    | ⟨ComputedElementConstructor⟩
    | ⟨ComputedAttributeConstructor⟩

Among the non-terminals on the right-hand-side, the mapping of ⟨XmlProcessingInstruction⟩ and ⟨XmlComment⟩ are very similar. We show the mapping of ⟨XmlComment⟩ only.

```
inp⟦⟨XMLComment⟩⟧ p = (⟨XMLComment⟩, p)
```

⟨ComputedDocumentConstructor⟩ ::= `document{` ⟨ExprSequence⟩ }

```
inp⟦⟨ComputedDocumentConstructor⟩⟧ p =
  (document { tau:copy-restricted-items(inp⟦⟨ExprSequence⟩⟧ p)}, p)
```

The translation of computed constructors is "copy-based" in in-place slicing. The result of a document constructor must be a well-formed document including only one root element, instead of a root element with a timestamp element. In addition, the evaluation of constructor in XQuery is copy-based (once an element is used to construct another node, its parent information in the original document is lost). Therefore, a copying approach is used here. The function `copy-restricted-items()` takes a sequence of items and their timestamps as inputs and copies the actual items with the correct timestamps without changing the structure of these items. This function is used in ⟨ComputedElementConstructor⟩ and ⟨ComputedAttributeConstructor⟩ as well.

⟨ComputedElementConstructor⟩ ::= `element` ⟨QName⟩ { ⟨ExprSequence⟩$^?$ }
    | `element` { ⟨Expr⟩} {⟨ExprSequence⟩$^?$}

```
inp⟦element ⟨QName⟩ {⟨ExprSequence⟩}⟧ p =
  (element ⟨QName⟩ {tau:copy-restricted-items(inp⟦⟨ExprSequence⟩⟧ p)},
   p)
inp⟦element ⟨Expr⟩ {⟨ExprSequence⟩}⟧ p =
  let $tau:s := inp⟦⟨Expr⟩⟧ p
  for $tau:p in tau:const-periods2(p, $tau:s) return
    (element {tau:copy-restricted-items(inp⟦⟨Expr⟩⟧ p)}
     {tau:copy-restricted-items(inp⟦⟨ExprSequence⟩⟧ p)},
     $tau:p)
```

⟨ComputedAttributeConstructor⟩ ::= `attribute` ⟨QName⟩ {⟨ExprSequence⟩$^?$}
    | `attribute` {⟨Expr⟩} {⟨ExprSequence⟩$^?$}

```
inp⟦⟨ComputedAttributeConstructor⟩⟧ p =
  let $tau:s := inp⟦⟨ExprSequence⟩⟧ p
  for $tau:p in tau:const-periods2(p, $tau:s) return
    (attribute ⟨QName⟩
     {tau:copy-restricted-items(inp⟦⟨ExprSequence⟩⟧ p)},
     $tau:p)
```

11. ⟨PathExpr⟩ ::= ⟨StepExpr⟩
⟨StepExpr⟩ ::= $⟨VarName⟩/⟨ForwardStep⟩ | $⟨VarName⟩/⟨ReverseStep⟩ | ⟨PrimaryExpr⟩
One major advantage of in-place slicing is that all the ⟨PathExpr⟩ can be handled. The reverse step is translated the same as the forward step. We show only the forward step here.

```
⟨ForwardAxis⟩ ::= child ::
                |descendant ::
                |attribute ::
                |self ::
                |descendant-or-self ::
                |following-sibling ::
                |following ::
                |namespace ::
⟨NodeTest⟩ ::= ⟨KindTest⟩ | ⟨NameTest⟩
```

inp ⟦$⟨VarName⟩/⟨ForwardStep⟩⟧ p =

```
  let $tau:s := tau:get-actual-items($⟨VarName⟩)
  let $tau:p := tau:get-periods($⟨VarName⟩)
  where tau:overlaps($tau:p, p) return
    let $tau:p1 := tau:intersection($tau:p, p)
    for $tau:step in $tau:s/⟨ForwardStep⟩
    where not(tau:special-node($tau:step))
          and tau:overlaps($tau:p1, $tau:step) return
      ($tau:step, tau:intersection($tau:p1, $tau:step))
```

The function `overlaps()` is used to examine if the two input parameters overlap in term of the valid-time. The function `intersection()` computes the valid-time intersection of the two input parameters.

The translation of the `attribute` axis requires more work, though it is similar to the above mapping. The reason is the representation of the time-varying attribute is an element.

inp ⟦$⟨VarName⟩/attribute::⟨NameTest⟩⟧ p =

```
  let $tau:s := tau:get-actual-items($⟨VarName⟩)
  let $tau:p := tau:get-periods($⟨VarName⟩)
  where tau:overlaps($tau:p, p) return
    let $tau:p1 := tau:intersection($tau:p, p) return
    (for $tau:a in $tau:s/attribute::⟨NameTest⟩ return
       ($tau:a, $tau:p1),
     for $tau:ta in $tau:s/timeVaryingAttribute[@name=⟨NameTest⟩]
        [@vtBegin<$tau:p1/@vtEnd][@vtEnd>$tau:p1/@vtBegin] return
       ($tau:ta, tau:period(min($tau:p1/@vtBegin, $ta/@vtBegin),
                            max($tau:p1/@vtEnd, $ta/@vtEnd))))
```

12. ⟨PrimaryExpr⟩ ::= ⟨Literal⟩
                | ⟨FunctionCall⟩
                | $⟨VarName⟩
                | (⟨ExprSequence⟩$^?$)

inp ⟦⟨Literal⟩⟧ p = (⟨Literal⟩, p)

inp ⟦$⟨VarName⟩⟧ p =

```
  tau:sequence-in-period($⟨VarName⟩, p)
```

13. $\langle$FunctionCall$\rangle$ ::= $\langle$QName$\rangle$ `(` ($\langle$Expr$_1\rangle$ (`,` $\langle$Expr$_2\rangle$)*)$^?$ `)`

    As in copy-based slicing, the user-defined functions and the built-in functions are treated differently. The mapping of user-defined function calls is as follows.

    $inp\,[\![\langle\text{FunctionCall}\rangle]\!]\; p = \langle\text{QName}\rangle$`(` ($inp\,[\![\langle\text{Expr}_1\rangle]\!]\; p$(`,` $inp\,[\![\langle\text{Expr}_2\rangle]\!]\; p$)*)$^?$ `)`

    The mapping built-in function can be written 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 actual items in each constant period are extracted from the input. The original function is called once on each constant period. Finally, the results are returned with their timestamps.

    $inp\,[\![\langle\text{FunctionCall}\rangle]\!]\; p =$

    ```
    let $tau:par1 := inp⟦⟨Expr1⟩⟧ p
    let $tau:par2 := inp⟦⟨Expr2⟩⟧ p
    for $tau:p in tau:all-const-periods2(p, union($tau:par1, $tau:par2))
    let $tau:s1 := tau:sequence-in-period($tau:par1, $tau:p)
    let $tau:s2 := tau:sequence-in-period($tau:par2, $tau:p) return
      (⟨QName⟩(tau:get-actual-items($tau:s1),
               tau:get-actual-items($tau:s2)), $tau:p)
    ```

    The function `all-const-periods2()` takes a time period as well as a sequence of items and their timestamps as inputs. It returns the constant periods of all the items and their descendants. The returned periods must be contained in the input period.

    Unlike copy-based slicing, in-place slicing can handle all the built-in functions since it does not copy the data until constructors are evaluated.

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

Using the in-place slicing, the normalized query shown in Figure 4 is mapped to the XQuery query in Figure 7. The translated result is longer than that of the copy-based slicing, because the actual items are extracted from the mixed sequence before each evaluation step and are paired with their timestamps after each evaluation step. However, it does not copy the nodes until the end of the evaluation. Hence, it does not necessarily take longer to run than the result of the copy-based slicing.

9.3 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. The example query discussed in last section is translated into the following query using the idiomatic slicing.

The auxiliary function `seq-path()` is defined in Appendix C. It returns the sequenced query results of a path expression. Compared with the query in Figure 5, the length of the query body is reduced dramatically by the idiomatic slicing.

Idiomatic slicing can also be used to eliminate some of the unneeded slicing. There are several situations in which idiomatic slicing applies. One is when a `let` expression binds a variable `$a` to a

```
import schema namespace rs="http://www.cs.arizona.edu/tau/RXSchema" at "RXSchema.xsd"
declare namespace tau = "www.cs.arizona.edu/tau/Func"
define function tau:const-periods2...
{-- validtime avg(for $c in --}
tau:apply-timestamp(
 let $tau:par1 :=
  (let $tau:s :=
    {-- let $tau:sequence:=document("CRM.xml") return --}
    (let $tau:s :=(let $tau:par2 :=("CRM.xml", tau:period("1000-01-01", "9999-12-31"))
                   for $tau:p in tau:all-const-periods2(tau:period("1000-01-01",
                                                        "9999-12-31"), $tau:par2)
                   let $tau:s1:= tau:sequence-in-period($tau:par2, $tau:p) return
                     (document(tau:get-actual-items($tau:s1)), $tau:p))
     for $tau:p in tau:const-periods2($tau:s)
     let $tau:sequence := tau:sequence-in-period($tau:s, $tau:p) return
      {-- for $tau:dot in $tau:sequence return --}
      let $tau:s1 := tau:sequence-in-period($tau:sequence, $tau:p)
      for $tau:v1 in $tau:s1
      let $tau:vi1 := index-of($tau:s1, $tau:v1)
      where ($tau:vi1 mod 2 = 1) return
       let $tau:p1 := item-at($tau:s1, $tau:vi1+1)
       let $tau:dot := ($tau:v1, $tau:p1) return
       {-- $tau:dot/descendant-or-self::customer) return --}
        let $tau:s2 := tau:get-actual-items($tau:dot)
        let $tau:p2 := tau:get-periods($tau:dot)
        where tau:overlaps($tau:p2, $tau:p1) return
         let $tau:p3 := tau:intersection($tau:p2, $tau:p1)
         for $tau:step in $tau:s2/descendant-or-self::customer
         where not(tau:special-node($tau:step)) and tau:overlaps($tau:p3,$tau:step)
         return ($tau:step, tau:intersection($tau:p3, $tau:step)))
   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 $c := ($tau:v, $tau:p) return
     {-- count(let $tau:sequence := $c return --}
     let $tau:par2 :=
      (let $tau:s1 := tau:sequence-in-period($c, $tau:p)
       for $tau:p1 in tau:const-periods2($tau:s1)
       let $tau:sequence := tau:sequence-in-period($tau:s1, $tau:p1) return
        {-- for $tau:dot in $tau:sequence return --}
        let $tau:s2 := tau:sequence-in-period($tau:sequence, $tau:p1)
        for $tau:v1 in $tau:s2
        let $tau:vi1 := index-of($tau:s1, $tau:v1)
        where ($tau:vi1 mod 2 = 1) return
         let $tau:p2 := item-at($tau:s1, $tau:vi1+1)
         let $tau:dot := ($tau:v1, $tau:p2) return
          {-- $tau:dot/child::supportIncident))) --}
          let $tau:s3 := tau:get-actual-items($tau:dot)
          let $tau:p3 := tau:get-periods($tau:dot)
          where tau:overlaps($tau:p3, $tau:p2) return
           let $tau:p4 := tau:intersection($tau:p3, $tau:p2)
           for $tau:step in $tau:s3/child::supportIncident
           where not(tau:special-node($tau:step)) and tau:overlaps($tau:p4,$tau:step)
           return ($tau:step, tau:intersection($tau:p1, $tau:step)))
     for $tau:p1 in tau:all-const-periods2($tau:p, $tau:par2)
     let $tau:s1 := tau:sequence-in-period($tau:par2, $tau:p1) return
      (count(tau:get-actual-items($tau:s1)), $tau:p1))
 for $tau:p in tau:all-const-periods2(tau:period("1000-01-01","9999-12-31"),$tau:par1)
 let $tau:s1 := tau:sequence-in-period($tau:par1, $tau:p) return
  avg(tau:get-actual-items($tau:s1)), $tau:p)
```

Figure 7: The Result of In-Place Per-Expression Slicing

```
{-- the original tauXquery:
    validtime avg(for $c in document("CRM.xml")//customer return
                      count($c/supportIncident)) --}
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"
define function tau:avg...
...
tau:avg(for $tau:i in tau:const-periods(tau:period("1000-01-01",
                                                   "9999-12-31"),
                   tau:seq-path(tau:period("1000-01-01", "9999-12-31"),
                      document("CRM.xml")//customer, document("CRM.xml")))
        for $tau:p in $tau:i/timestamp
        let $c := tau:copy-restricted-subtree($tau:p, $tau:i) return
         tau:count(tau:seq-path($tau:p, $c/supportIncident, $c)))
```

Figure 8: The Result of Idiomatic Slicing

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 `copy-restricted-subtree()` will do useless work on `$a`.

9.4 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.

10 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.

SQL/Temporal [SBJ98] is a query language obtained by adding valid-time support to SQL3. The classification of temporal queries to current query, sequenced query, and representational query was introduced in this language. We use this classification in the language design of τXQuery. Torp et al. Proposed the layered strategy to implement temporal DBMS [TJB97]. The intension is to maximally reuse the facilities of an existing SQL implementaion. We adopt this strategy for the similar consideration. The mapping of τXQuery to XQuery is quite different from the mapping of SQL/Temporal to SQL due to the difference between the underlying data models and the base languages.

As mentioned in Section 1, there has been some work addressing the transaction time dimension of XML. These papers focus on XML versioning including representing, detecting, and querying the changes in XML documents. Our work concentrates on how to evaluateτXQuery 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 . This work can be viewed as an extension to a conventional semistructured database. Temporal constituents in XML and their representaion were investigated by Manukyan et al. [MK01]. They did not address the problem of querying temporal XML. Cao et.al. proposed a data model for warehousing historical web information [CLN00]. Their method is used to request web pages in historical warehouse of web pages. They did not disscuss querying the data in each page.

Grandi and Mandreoli [GM99] 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 [BKT02]. 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.

11 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.

12 Acknowledgements

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

[Ahlert00] H. Ahlert, "Enterprise Customer Management: Integrating Corporate and Customer Information," in **Relationship Marketing**, Springer, 2000.

[AP02] J. Anton and N. L. Petouhoff, **Customer Relationship Management**, Prentice Hall, 2002.

[BBJS97] Bair, J., M. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Notions of Upward Compatibility of Temporal Query Languages," *Business Informatics (Wirtschafts Informatik)*, Vol. 39, No. 1, February 1997, pp. 25–34.

[BJS00] Böhlen, M., C. S. Jensen, and R. T. Snodgrass, "Temporal Statement Modifiers," *ACM Transactions on Database Systems*, 25(4):407–456. December, 2000.

[BSS96] 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.

[BKT02] 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.

[CLN00] Y. Cao, E. P. Lim, and W. K. Ng, "Storage Management of a Historical Web Warehousing System," in *Proceedings of the International Conference on DEXA*, pp. 457–466. London, UK, September 2000.

[CTZ00] S-Y. Chien, V. J. Tsotras, and C. Zaniolo, "A Comparative Study of Version Management Schemes for XML Documents," TIMECENTER Technical Report TR-51, TimeCenter, 2000.

[CTZ01] S-Y. Chien, V. J. Tsotras, and C. Zaniolo, "Copy-based Versus Edit-based Version Management Schemes for Structured Documents," in *Proceedings of International Workshop on RIDE*, pp. 95-102. Heidelberg, Germany, April 2001.

[ZCT01] S-Y. Chien, V. J. Tsotras, and C. Zaniolo, "Efficient Management of Multiversion Documents by Object Referencing," in *Proceedings of the International Conference on Very Large Databases*, pp. 291-300. Rome, Italy, September, 2001.

[CTZ02] 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.

[CTZ$^+$02] 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.

[CAM02] 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.

[CCD02] F. Currim, S. Currim, C. E. Dyreson, and R. T. Snodgrass, “Temporal XML Schema,” March 2003, in preparation.

[Dyr03] C. E. Dyreson, ”Temporal Coalescing with Now, Granularity, and Incomplete Information,” in *Proceedings of the ACM SIGMOD International Conference*, San Diego, CA, June 2003.

[DBJ02] 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.

[GPT03] S. Gallant, G. Piatetsky-Shapiro, and M. Tan, ”Value-based Data Mining for CRM,” in Proceedings of the SIGKDD International Conference, 2003.

[GM99] 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.

[IBM02] IBM, “Xperanto Technology Demo,” Mar 2002. `http://www7b.boulder.ibm.com/dmdd/library/demos/0203xperanto/0203xperanto.html`.

[JD98] C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, “A Consensus Glossary of Temporal Database Concepts—February 1998 Version,” in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, andS. Sripada (eds.), Springer-Verlag, pp. 367–405, 1998.

[LM97] N. A. Lorentzos and Y. G. Mitsopoulos, “SQL Extension for Interval Data,” *IEEE Transactions on Knowledge and Data Engineering* 9(3): 480–499, 1997.

[Luc03] Lucent-Bell Lab and AT&T Research, “Galax Version 0.3.0,” `http://db.bell-labs.com/galax/`

[MK01] M. G. Manukyan and L. A. Kalinichenko, “Temporal XML,” in *Proceedings of ADBIS*, Vilnius, Lithuania, September 2001.

[MAC$^+$01] 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.

[MS02] Microsoft Corporation, “XML Query Language Demo,” `http://131.107.228.20/xquerydemo`

[ORA02] Oracle Corporation, “Oracle XQuery Prototype: Querying XML the XQuery way,” March 2002. `http://otn.oracle.com/sample_code/tech/xml/xmldb/xmldb_xquerydownload.html`.

[SA86] R. T. Snodgrass and I. Ahn, “Temporal Databases.” *IEEE Computer*, 19(9):35–42, September 1986.

[SBJ98] R. T. Snodgrass, M. H. Bohlen, C. S. Jensen, and A. Steiner, “Transitioning Temporal Support in TSQL2 to SQL3”. In O. Etzion, S. Jajodia, and S. M. Sripada, editors, **Temporal Databases: Research and Practice**, volume 1399 of *Lecture Notes in Computer Science*, pp. 150–194, 1998

[SGM93] 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)

[Snod87] Richard T. Snodgrass, “The Temporal Query Language TQuel”. *ACM Transactions on Database Systems* 12(2): 247-298 (1987)

[Stoy79] J. E. Stoy, **Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory**. The MIT Press 1979.

[Tom98] D. Toman, "Point-based temporal extensions of SQL and their efficient implementation". In O. Etzion, S. Jajodia, and S. M. Sripada, editors, **Temporal Databases: Research and Practice**, volume 1399 of *Lecture Notes in Computer Science*, pp. 211–237. Springer, 1998.

[TJB97] K. Torp, C. S. Jensen, and M. Bohlen, "Layered Temporal DBMS's–Concepts and Techniques," in *Proceedings of International Conference on Database Systems for Advanced Applications*, Melbourne, Australia, April 1997.

[W3C01] World Wide Web Consortium, "XML Schema Part 0: Primer," *W3C Recommendation*, May, 2001. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502`

[W3C02] 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/`

[W3C02] 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/`

[XBench] `http://db.uwaterloo.ca/∼ddbms/projects/xbench`

[XMach] `http://dbs.uni-leipzig.de/en/projekte/XML /XmlBenchmarking.html`

[XMark] `http://www.xml-benchmark.org`

[XOO7] `http://www.comp.nus.edu.sg/∼ebh/XOO7.html`

A Schema for Valid Timestamp: RXSchema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.cs.arizona.edu/tau/RXSchema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:rs="http://www.cs.arizona.edu/tau/RXSchema" elementFormDefault="qualified"
    attributeFormDefault="unqualified" version="October, 2002">

  <xs:annotation>
    <xs:documentation>
       XML Schema file for describing the Representational Schema.
       Definitions for validtime type, element timestamps datatypes, and time-varying
       attribute data type.
    </xs:documentation>
  </xs:annotation>

  <xs:simpleType name="validTimeType">
    <xs:union memberTypes="xs:dateTime xs:date rs:foreverType" />
  </xs:simpleType>

  <xs:simpleType name="foreverType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="forever"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="vtStep">
    <xs:attribute name="vtBegin" type="rs:validTimeType"/>
  </xs:complexType>

  <xs:complexType name="vtExtent">
    <xs:complexContent>
      <xs:extension base="rs:vtStep">
        <xs:attribute name="vtEnd" type="rs:validTimeType"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="vtAttributeTS">
    <xs:complexContent>
      <xs:extension base="rs:vtExtent">
        <xs:attribute name="name" type="xs:string"/>
        <xs:attribute name="value" type="xs:string"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

B Schema for Time-Varying Value: Tvv.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.cs.arizona.edu/tau/Tvv"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:rs="http://www.cs.arizona.edu/tau/RXSchema" elementFormDefault="qualified"
  attributeFormDefault="unqualified" version="October, 2002">
  <xs:annotation>
    <xs:documentation>
```

```
        XML Schema file defining the type for time-varying simple value
     </xs:documentation>
  </xs:annotation>
  <xs:import namespace="http://www.cs.arizona.edu/tau/RXSchema"
    schemaLocation="RXSchemaTest.xsd"/>

  <xs:complexType name="timeVaryingValueType">
    <xs:sequence>
      <xs:element name="timestamp" type="rs:vtExtent"/>
      <xs:element name="value" type="xs:AnyType"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

C Auxiliary Functions

Here we provide an implementation of all of the auxiliary functions used in Sections 5 and 9. They are given in alphabetical order.

- Function `tau:all-const-periods()`
  This function takes a time period as well as a list of nodes and computes all the periods during which no single value in any of the nodes changes. It is used in maximally-fragmented slicing to find the constant periods in the input documents and in mapping built-in function in copy-based per-expression slicing.

  ```
  define function tau:all-const-periods(rs:vtExtent $p, xsd:node* $src)
  returns rs:vtExtent*
  {
    {-- get all the time points and sort the list without duplicates --}
    let $ts := distinct-values( for $doc in $src
                 for $t in tau:all-time-points($doc, $p/@vtBegin, $p/@vtEnd)
                 order by $t
                 return $t )
    for $index in (1 to count($ts)-1)
    let $pbt := item-at($ts, $index)
    let $pet := item-at($ts, $index+1)
    return <timestamp vtBegin="{$pbt}" vtEnd="{$pet}"/>
  }
  ```

- Function `tau:all-const-periods2()`
  This function takes a time period as well as a sequence of items and their timestamps as inputs. It computes all the periods during which no single value in any of the items changes. The returned periods must be contained in the input period. It is used in mapping the built-in functions in in-place per-expression slicing.

  ```
  define function tau:all-const-periods2(rs:vtExtent $p, item* $src)
  returns rs:vtExtent*
  {
    {-- get all the time points and sort the list without duplicates --}
    let $ts := distinct-values( for $doc in $src
                 where index-of($src, $doc) mod 2 = 1 return
                   let $dp := item-at($src, index-of($src, $doc) + 1)
                   where tau:overlaps($dp, $p) return
                     let $newp := tau:intersection($dp, $p)
                     for $t in tau:all-time-points($doc, $newp/@vtBegin,
                                                         $newp/@vtEnd)
                     order by $t
                     return $t )
  ```

```
  for $index in (1 to count($ts)-1)
  let $pbt := item-at($ts, $index)
  let $pet := item-at($ts, $index+1)
  return <timestamp vtBegin="{$pbt}" vtEnd="{$pet}"/>
}
```

- Function `tau:all-time-points()`
  This function takes a sequence of nodes and a time period (represented as two `dateTime` value) and returns the time points when the state of the input nodes or their descendants are changed. The returned time points must be contained in the input period. It is called by `tau:all-const-periods()` and `tau:all-const-periods2()`.

  ```
  define function tau:all-time-points(xsd:node* $src, xs:dateTime $bt,
  xs:dateTime $et) returns xs:dateTime*
  {
    for $i in $src
    for $e in $i/*
    return if (name($e) = "timestamp") or (name($e) = "timeVaryingAttribute")
           then {-- timestamp subelement or time-varying attribut--}
             for $t in ($e/@vtBegin, $e/@vtEnd)
             where ($bt <= $t) and ($t < $et)
             return data($t)
           else {-- find the time-points recursively --}
             tau:all-time-points($e, $bt, $et)
  }
  ```

- Function `tau:apply-timestamp()`
  This function takes a sequence and makes a copy of the items in the odd positions with the correct timestamps computed according to the timestamp that follows it immediately. It is used only in the in-place per-expression slicing to compute the final result of the query.

  ```
  define function tau:apply-timestamp(item* $src) return item*
  {
    for $v in $src
    let $vi := index-of($src, $v)
    where ($vi mod 2 = 1) return
      tau:copy-restricted-subtree(item-at($src, $vi+1), $v)
  }
  ```

- Function `tau:associate-timestamp()`
  This function 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. It is used in the maximally-fragmented slicing to compute the final result of the query.

  ```
  define function tau:associate-timestamp(rs:vtExtent $p, item* $src)
  returns xsd:node*
  {
    for $i in $src
    return typeswitch ($i)
            case xs:document return
              document
              {
                tau:associate-timestamp($p, $i/child::node())
              }
            case xs:element return
              {-- the item is an element --}
              element node-name($i) {
                for $a in $i/@*
                return attribute node-name($a) {$a} ,
                {$p},
  ```

```
              $i/child::node()
            }
          case xs:attribute return
            {-- the item is an attribute --}
            element timeVaryingAttribute {
              attribute name {node-name($i)},
              attribute value {xf:data($i)},
              attribute vtBegin {$p/@vtBegin},
              attribute vtEnd {$p/@vtEnd}
            }
          case atomic value return
            {-- the item is an atomic value --}
            <timeVaryingValue>
              {$p}
              <value>{$i}</value>
            </timeVaryingValue>
          default return $e
}
```

- Function `tau:const-periods()`
  This function takes a time period and a sequence of nodes and returns the constant periods for each of the nodes in the input sequence, not for all the subelements. It is used in mapping almost every expression in the copy-based per-expression slicing.

```
define function tau:const-periods(rs:vtExtent $p, xsd:node* $src)
returns rs:vtExtent*
{
  {-- get all the time points and sort the list without duplicates --}
  let $ts := distinct-values( for $doc in $src
               for $t in tau:time-points($doc, $p/@vtBegin, $p/@vtEnd)
               order by $t
               return $t )
  for $index in (1 to count($ts)-1)
  let $pbt := item-at($ts, $index)
  let $pet := item-at($ts, $index+1)
  return <timestamp vtBegin="{$pbt}" vtEnd="{$pet}"/>
}
```

- Function `tau:const-periods2()`
  This function takes a sequence, including items and their timestamps, and a period as inputs. It returns the constant periods of this sequence of items contained in the input period. It used in mapping almost every expression in the in-place per-expression slicing.

```
define function tau:const-periods2(rs:vtExtent $p, item* $src)
return rs:vtExtent*
{
  {-- get all the time points and sort the list without duplicates --}
  let $ts := distinct-values(
               for $t in tau:time-points2($src, $p/@vtBegin, $p/@vtEnd)
               order by $t
               return $t )
  for $index in (1 to count($ts)-1)
  let $pbt := item-at($ts, $index)
  let $pet := item-at($ts, $index+1)
  return <timestamp vtBegin="{$pbt}" vtEnd="{$pet}"/>
}
```

- Function `tau:copy-restricted-items`
  This function takes a sequence of items and their timestamps as inputs and copies the actual items

with the correct timestamps without changing the structure of these items. It is used in mapping computed constructors in in-place per-expression slicing.

```
define function tau:copy-restricted-items(item* $e, rs:vtExtent* $p)
returns item*
{
  for $i in $e
  return
    typeswitch ($i)
    case xs:document return
      document{
        tau:copy-restricted-items($i/child::node(), $p)
      }
    case rs:vtExtent return ()
    case rs:attribTS return
      for $per in $p
      return
        if ($i/@vtBegin < $per/@vtEnd) and ($i/@vtEnd > $per/@vtBegin)
        then element timeVaryingAttribute {
               attribute name {$i/@name},
               attribute value {$i/@value},
               attribute vtBegin {max($i/@vtBegin, $per/@vtBegin)},
               attribute vtEnd {min($i/@vtEnd, $per/@vtEnd)}
             }
    case xs:element return
      let $localps := $i/timestamp return
        if (empty($localps))
        then let $currentps := $p return
               element node-name($i) {
                 (for $a in $i/@* return
                    tau:copy-restricted-items($currentps, $a),
                  if (empty($i/*))
                  then data($i)
                  else for $c in $i/child::node() return
                         tau:copy-restricted-items($currentps, $c))
               }
        else let $currentps := tau:time-intersection($localps, $p)
             where not empty($currentps) return
               element node-name($i) {
                 (for $a in $i/@* return
                    tau:copy-restricted-items($currentps, $a),
                  for $ps in $currentps return $ps,
                  for $c in $i/child::node() return
                    tau:copy-restricted-items($currentps, $c))
               }
    default return $i
}
```

- Function `tau:copy-restricted-subtree`
  This function takes one or more time periods and a variable as input parameters. It makes a copy of the input variable and removes the descendants that are not valid in the input periods. It is used frequently in copy-based per-expression slicing and is also used to compute the final result of the query in in-place per-expression slicing.

```
define function tau:copy-restricted-subtree(rs:vtExtent* $p, xs:node* $e)
returns xs:node*
{
  for $i in $e
  return
    typeswitch ($i)
    case xs:document return
      document{
        tau:copy-restricted-subtree($p, $i/child::node())
      }
    case rs:vtExtent return ()
    case rs:attribTS return
      for $per in $p
      return
        if ($i/@vtBegin < $per/@vtEnd) and ($i/@vtEnd > $per/@vtBegin)
        then element timeVaryingAttribute {
               attribute name {$i/@name},
               attribute value {$i/@value},
               attribute vtBegin {max($i/@vtBegin, $per/@vtBegin)},
               attribute vtEnd {min($i/@vtEnd, $per/@vtEnd)}
             }
    case xs:element return
      let $localps := $i/timestamp
      let $currentps := (if empty($localps)
                         then $p
                         else tau:time-intersection($localps, $p))
      where not empty($currentps)
      return element node-name($i) {
               for $a in $i/@*
               return tau:copy-restricted-subtree($currentps, $a),

               if xf:empty($i/*)
               then <value>xf:data($i)</value>
               else
                 for $c in $i/child::node()
                 return
                   if (node-name($c) = "value")
                   then $c
                   else tau:copy-restricted-subtree($currentps, $c),

               for $ps in $currentps
               return $ps
             }
    case xs:attribute return
      for $per in $p
      return element timeVaryingAttribute {
               attribute name {node-name($i)},
               attribute value {xf:data($i)},
               attribute vtBegin {$per/@vtBegin},
               attribute vtEnd {$per/@vtEnd}
             }
    default return $i
}
```

- Function `tau:element-const-periods()`
  This function takes a sequence of documents and a sequence of strings representing 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. It is used in the selected node slicing.

```
define function tau:element-const-periods(rs:vtExtent $p, xsd:node* $src,
item*$nodes) returns rs:vtExtent*
{
  {-- get all the time points and sort the list without duplicates --}
  let $ts := distinct-values( for $doc in $src
                for $t in tau:element-time-points($doc, $nodes, $p/@vtBegin,
                                                                    $p/@vtEnd)
                order by $t
                return $t )
  for $index in (1 to count($ts)-1)
  let $pbt := item-at($ts, $index)
  let $pet := item-at($ts, $index+1)
  return <timestamp vtBegin="{$pbt}" vtEnd="{$pet}"/>
}
```

- Function `tau:element-time-points()`
  This function takes a sequence of documents, a sequence of strings representing node names (elements or attributes), and a time period represented by two `dateTime` value. It collects the begin and end time of the nodes whose name appears in the input sequence. The returned time points must be contained in the input period. It is called only by the function `tau:element-const-periods()`.

```
define function tau:element-time-points(xsd:node* $src, item* $nodes
xs:dateTime $bt, xs:dateTime $et) returns xs:dateTime*
{
  for $i in $src
  for $e in $i/* return
     if ((name($e) = "timestamp" and name($i) = $nodes) or
         (name($e) = "timeVaryingAttribute" and $e/@name = $nodes))
     then {-- timestamp subelement or time-varying attribute --}
       for $t in ($e/@vtBegin, $e/@vtEnd)
       where ($bt <= $t) and ($t < $et)
       return data($t)
     else {-- find the time-points recursively --}
       tau:element-time-points($e, $nodes, $bt, $et)
}
```

- Function `tau:get-actual-items()`
  This function takes a sequence and returns only the items in the odd position. It is used in in-place per-expression slicing to seperate the items from their timestamps.

```
define function tau:get-actual-items(item* $src) return item*
{
  for $v in $src
  where index($src, $v) mod 2 = 1 return
    $v
}
```

- Function `tau:get-periods()`
  This function takes a sequence and returns the timestamps in the even position as a sequence. It is used in the in-place per-expression slicing to separate the timestamps from their items.

```
define function tau:get-periods(item* $src) return rs:vtExtent*
{
  for $p in $src
  where index($src, $p) mod 2 = 0 return
    $p
}
```

- Function `tau:interleave()`
  This function takes two sequences as inputs and interleaves them as one sequence. It is used in the in-place per-expression slicing to combine the items with their timestamps.

```
define function tau:interleave(item* $src, rs:vtExtent $ps) return item*
{
  for $i in (1 to count($src))
  let $v := item-at($src, $i)
  let $p := item-at($ps, $i) return
    ($v, $p)
}
```

- Function `tau:intersection()`
  This function computes the valid-time intersection of the two input parameters. It is used in in-place per-expression slicing.

```
define function tau:intersection(item $src1, item $src2)
return rs:vtExtent
{
  let $p1 := typeswitch ($src1)
               case rs:vtExtent $p return $p
               default $e return
                 if (empty($e/timestamp))
                 then tau:period("1000-01-01", "9999-12-31")
                 else $e/timestamp
  let $p2 := typeswitch ($src2)
               case rs:vtExtent $p return $p
               default $e return
                 if (empty($e/timestamp))
                 then tau:period("1000-01-01", "9999-12-31")
                 else $e/timestamp
  return tau:time-intersection($p1, $p2)
}
```

- Function `tau:overlaps()`
  The function `overlaps()` is used to examine if the two input parameters overlap in term of the valid-time. It is used in in-place per-expression slicing.

```
define function tau:overlaps(item $src1, item $src2) return xs:boolean
{
  let $p1 := typeswitch ($src1)
               case rs:vtExtent $p return $p
               default $e return
                 if (empty($e/timestamp))
                 then tau:period("1000-01-01", "9999-12-31")
                 else $e/timestamp
  let $p2 := typeswitch ($src2)
               case rs:vtExtent $p return $p
               default $e return
                 if (empty($e/timestamp))
                 then tau:period("1000-01-01", "9999-12-31")
                 else $e/timestamp
  return
   if (empty(tau:time-intersection($p1, $p2))
   then false
   else true
}
```

- Function `tau:period()`
  This function takes two `dateTime` value as begin time and end time and constructs a period represented by an element of the type `rs:vtExtent`. It is used in all the slicing approaches.

```
define function tau:period(xs:dateTime $bt, xs:dateTime $et)
returns rs:vtExtent
{
  <timestamp vtBegin="{$bt}" vtEnd="{$et}"/>
}
```

- Function `tau:periods-of()`
  This function returns all the timestamps associated with the input node. It is used in mapping the ⟨ForExpr⟩ in copy-based per-expression slicing.

```
define function tau:periods-of(item $e) returns rs:vtExtent*
{
  $e/timestamp
}
```

- Function `tau:seq-path()`
  This function takes a time period, a sequence of nodes as the result of evaluating a path expression, and a context node of the path expression as inputs. It returns the sequenced query results of a path expression. It is used in idiomatic slicing.

```
define function seq-path(rs:vtExtent $p, node* $TXQ, node $e)
returns xs:node*
{
  typeswitch ($e)
    case xs:document return
      tau:seq-path($p, $TXQ, $e/child::node())
    case xs:attribTS return
      {-- time-varying attribute --}
      if some $i in $TXQ satisfies ($i is $e))
      then let $ap := tau:period($e/@vtBegin, $e/@vtEnd)
           let $cp := tau:time-intersection($p, $ap)
           where not empty($cp) return
             element timeVaryingAttribute {
               attribute name {$e/@name},
               attribute value {$e/@value},
               attribute vtBegin {$cp/@vtBegin},
               attribute vtEnd {$cp/@vtEnd}
             }
      else ()
    case xs:vtExtent return ()
    case xs:attribute return
      if some $i in $TXQ satisfies ($i is $e))
      then element timeVaryingAttribute {
             attribute name {node-name($e)},
             attribute value {xf:data($e)},
             attribute vtBegin {$p/@vtBegin},
             attribute vtEnd {$p/@vtEnd}
           }
      else ()
    case xs:element return
      let $localps := $e/timestamp
      let $currentps := (if empty($localps)
                         then $p
                         else tau:time-intersection($localps, $p))
      where not empty($currentps) return
        if some $i in $TXQ satisfies ($i is $e)
        then tau:copy-restricted-subtree($currentps, $e)
        else for $eachp in $currentps
             for $c in $e/child::node() return
             tau:seq-path($eachp, $TXQ, $c)
    default return
      if some $i in $TXQ satisfies ($i is $e)
      then $e
      else ()
}
```

- Function `tau:sequence-in-period()`
  This function takes two input parameters, a sequence of items with their timestamps and a period. It computes the overlap of the valid period of each item and the input period. Those items that are not valid in the input period are filtered out. The rest items with the overlapped periods are returned in a sequence. It is used in the in-place per-expression slicing.

```
define function tau:sequence-in-period()(item* $src, rs:vtExtent $p)
return item*
{
  for $v in $src
  where index-of($src, $v) mod 2 = 1
  let $p1 := item-at($src, index-of($src, $v)+1)
  where tau:overlaps($p1, $p) return
    ($v, tau:intersection($p1, $p))
}
```

- Function `tau:snapshot()`
  This function takes an item n and a time t as the input parameters and returns the snapshot of n at time t. This snapshot item has no valid timestamps; elements not valid now have been stripped out. It is used in current query, maximally-fragmented slicing, and copy-based per-expression slicing.

```
define function tau:snapshot(xsd:item $e, xs:dateTime $time)
returns xsd:item
{
  typeswitch $e
  case document return
    {-- $e is a document node --}
    document
    {
      if (node-name($e/*[1]) = "valueVaryingRoot")
      then for $r in $e/valueVaryingRoot/child::node()
           return tau:snapshot($r, $time)
      else for $r in $e/child::node()
           return tau:snapshot($r, $time)
    }

  case processing-instruction return $e
  case comment return $e
  case text return $e
  case atomic value* return $e
    {-- the above four types don't have valid timestamps --}

  case element of type tvv:timeVaryingValueType return
    if (every $vt in $e/timestamp satisfies ($vt/@vtBegin > $time or
            $vt/@vtEnd <= time))
    then ()
    else xf:data($e/value)
  case element return
    if (not(empty($e/timestamp)) and (every $vt in $e/timestamp satisfies
                          ($vt/@vtBegin > $time or $vt/@vtEnd <= time)))
    then ()
      {-- if $e time varying and its valid time period does not
          contain $time --}
    else element {node-name($e)}
      {
        {-- return non-temporal attributes --}
       (for $a in $e/@*
        return attribute {node-name($a)} {$a},

        {-- return the value of the element if it has no subelement --}
        if empty($e/*)
        then xf:data($e)
        else {-- return time-varying attributes that are valid at $time --}
         (for $ta in $e/timeVaryingAttribute
          where ($ta/@vtBegin <= $time and $ta/@vtEnd > $time)
          return attribute {$ta/@name} {$ta/@value},

          {-- return the snapshot of all the subelements
              except for the timestamps--}
          for $s in $e/child::node()
          where (node-name($s) != "timestamp" and
                 node-name($s) != "timeVaryingAttribute")
          return if node-name($s) = "value"
                 then xf:data($s)
                 else tau:snapshot($s,$time)))
      }
}
```

- Function `tau:special-node()`
  This function returns true when the input node is a special node (e.g., `timestamp` and `time-VaryingAttribute`) for representing the valid periods. It is used in per-expression slicing.

```
define function tau:special-node(xsd:node $src) return xs:boolean
{
  if (local-name($src) = "timestamp" or
      local-name($src) = "timeVaryingAttribute or
      local-name($src) = "value")
  then true
  else false
}
```

- Function `tau:time-intersection` This function takes two sequences of timestamps and returns the intersections between the periods in one sequence and the periods in the other sequence. It is called by several other auxiliary functions.

```
define function tau:time-intersection(rs:vtExtent* $localps,
rs:vtExtent* $ps) returns rs:vtExtent*
{
  for $lp in $localps
  for $p in $ps
  where ($lp/@vtBegin < $p/$vtEnd) and ($lp/@vtEnd > $p/$vtBegin)
  return element timestamp {
           attribute vtBegin {max($lp/@vtBegin, $p/@vtBegin)},
           attribute vtEnd {min($lp/@vtEnd, $p/@vtEnd)}
         }
}
```

- Function `tau:time-points()`
  This function takes a sequence of nodes and a time period (represented as two `dateTime` value) and returns the begin and end time of the input nodes (not including the sub-elements). The returned time points must be contained in the input period. It is called only by `const-periods()`.

```
define function tau:time-points(xsd:node $src, xs:dateTime $bt,
xs:dateTime $et) returns xs:dateTime*
{
  for $e in $src/timestamp
  for $t in ($e/@vtBegin, $e/@vtEnd)
  where ($bt <= $t) and ($t < $et)
  return $t
}
```

- Function `tau:time-points2()`
  This function takes a sequence of items and their timestamps, and a time period (represented as two `dateTime` value) as the inputs. It returns the begin and end time of the input timestamps. The returned time points must be contained in the input period. It is called only by `tau:const-periods2()`.

  ```
  define function tau:time-points2(item* $src, xs:dateTime $bt,
  xs:dateTime $et) returns xs:dateTime*
  {
    for $e in $src
    where index-of($src, $e) mod 2 = 0 return
      (if ($e/@vtBegin >= $bt and $e/@vtBegin <$et)
       then $e/@vtBegin
       else (),
       if ($e/@vtEnd >= $bt and $e/@vtEnd <$et)
       then $e/@vtEnd
       else ())
  }
  ```

D Non-Temporal Schema: CRM.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.cs.arizona.edu/stratum/CRM"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:annotation>
    <xs:documentation>
       Non-temporal schema for customer relationship management.
    </xs:documentation>
  </xs:annotation>

  <xs:element name="CRMdata">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="customer">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element ref="contactInfo"/>
        <xs:element ref="directedPromotion" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="supportIncident" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="supportLevel" type="slType"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="contactInfo">
    <!-- Definition of subelements of contactInfo includes name, address, and phone. -->
  </xs:element>

  <!-- Definition of directedPromotion -->

  <xs:element name="supportIncident">
    <xs:complexType mixed="false">
```

```
      <xs:sequence>
        <xs:element name="product" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:element ref="action" minOccurs="0" maxOccurs="unbounded/>
        <xs:element ref="resolution"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="action">
    <!-- Definition of subelements of action includes who, what, and handoff. -->
  </xs:element>

  <!-- Definition of resolution -->

  <xs:simpleType name="slType">
    <xs:restriction base="xs:string">
      <xs:pattern value="platinum|gold|silver|regular"/>
    </xs:restriction>
  </xs:simpleType>

<xs:schema>
```

E Temporal Annotations on the CRM Schema: CRM.tsd

The temporal annotation specifies which nodes are time-varying, whether they are value-varying or existance-varying, whether they are event data or state data, and whether they change over valid time, transaction time, or both. In this example, We annotate three elements and one attribute to be time-varying in term of valid time. They are all state data. Two of them are existance-varying, the others are value-varying.

```
<?xml version="1.0" encoding="UTF-8"?>
<temporalAnnotatedSchema xmlns="http://www.cs.arizona.edu/tau/TXSchema"
     xmln:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.cs.arizona.edu/tau/TXSchema TXSchema.xsd"/>
  <nonTemporalSchema schemaLocation="http://www.cs.arizona.edu/CRM.xsd"/>
  <validTime target="/CRMdata/customer/contactInfo" kind="state"/>
  <validTime target="/CRMdata/customer/@supportLevel" kind="state"/>
  <validTime target="/CRMdata/customer/supportIncident" kind="state"
             existanceVarying="true"/>
  <validTime target="/CRMdata/customer/supportIncident/action" kind="state"
             existanceVarying="true"/>
</temporalAnnotatedSchema>
```

F Physical Annotations on the CRM Schema

The physical annotation indicates which nodes are physically timestamped and what data type is used to represent the timestamps. It is independent from the temporal annotation. Given a non-temporal schema and a temporal annotation, multiple physical annotation strategies can be developed. Here, we provide two different physical annotations for the CRM example.

F.1 Physical Annotations with Timestamps at The Same Level as Temporal Annotaions: CRM1.psd

In this example, we annotate the schema with timestamps at the same level as the temporal annotations. The timestamp type is extent, which is a period represented by begin time and end time.

```
<?xml version="1.0" encoding="UTF-8"?>
<physicalAnnotatedSchema xmlns="http://www.cs.arizona.edu/tau/PXSchema"
     xmln:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.cs.arizona.edu/tau/PXSchema PXSchema.xsd"/>
  <temporalAnnotatedSchema schemaLocation="http://www.cs.arizona.edu/CRM.tsd"/>
  <validTime target="/CRMdata/customer/contactInfo" timeStampType="extent"/>
  <validTime target="/CRMdata/customer/@supportLevel" timeStampType="extent"/>
  <validTime target="/CRMdata/customer/supportIncident" timeStampType="extent"/>
  <validTime target="/CRMdata/customer/supportIncident/action" timeStampType="extent"/>
</physicalAnnotatedSchema>
```

F.2 Physical Annotations with Timestamps at Root: CRM2.psd

In this example, we annotate only the root node of the schema with timestamps. Whenever a single value changes in the document, a new copy of the whole tree is created.

```
<?xml version="1.0" encoding="UTF-8"?>
<physicalAnnotatedSchema xmlns="http://www.cs.arizona.edu/tau/PXSchema"
     xmln:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.cs.arizona.edu/tau/PXSchema PXSchema.xsd"/>
  <temporalAnnotatedSchema schemaLocation="http://www.cs.arizona.edu/CRM.tsd"/>
  <validTime target="/CRMdata" timeStampType="extent"/>
</physicalAnnotatedSchema>
```

G Representational Schema for the CRM Example

The non-temporal schema, the temporal annotation, and the physical annotation imply a representational schema, which defines the structure of the temporal XML documents and the data type of each element and attribute. Again, we show the two different representational schemas that result from the two different physical annotations for the CRM example.

G.1 Representational Schema for Physical Annotations in CRM1.psd: repCRM1.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:rs="http://www.cs.arizona.edu/tau/RXSchema"
           elementFormDefault="qualified" attributeFormDefault="unqualified">

  <xs:import namespace="http://www.cs.arizona.edu/tau/RXSchema"
             schemaLocation="RXSchema.xsd"/>

  <xs:element name="CRMdata">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="customer">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element ref="contactInfo" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="directedPromotion" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="supportIncident" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="rs:timeVaryingAttribute" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!-- Definition of contactInfo -->
  <!-- Definition of directedPromotion -->

  <xs:element name="supportIncident">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element name="product" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:element ref="action" minOccurs="0" maxOccurs="unbounded/>
        <xs:element ref="resolution"/>
        <xs:element ref="rs:timestamp"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!-- Definition of action -->
  <!-- Definition of resolution -->

</xs:schema>
```

G.2 Representational Schema for Physical Annotations in CRM2.psd: repCRM2.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:rs="http://www.cs.arizona.edu/tau/RXSchema"
           elementFormDefault="qualified" attributeFormDefault="unqualified">

  <xs:import namespace="http://www.cs.arizona.edu/tau/RXSchema"
             schemaLocation="RXSchema.xsd"/>

  <xs:element name="valueVaryingRoot">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="CRMdata" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="CRMdata">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="rs:timestamp"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="customer">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element ref="contactInfo"/>
        <xs:element ref="directedPromotion" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="supportIncident" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="supportLevel" type="slType"/>
    </xs:complexType>
  </xs:element>

  <!-- Definition of contactInfo -->
  <!-- Definition of directedPromotion -->

  <xs:element name="supportIncident">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element name="product" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:element ref="action" minOccurs="0" maxOccurs="unbounded/>
        <xs:element ref="resolution"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!-- Definition of action -->
  <!-- Definition of resolution -->

  <xs:simpleType name="slType">
    <xs:restriction base="xs:string">
      <xs:pattern value="platinum|gold|silver|regular"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

H Example Instances

In this section, we give two example instances defined by the two representational schemas. They are different in that the timestamps are placed in different levels. In `CRM1.xml`, timestamps are placed in three different levels of the tree, while in `CRM2.xml`, timestamps are placed only at the surrogate root element. These two temporal XML documents are snapshot equivalent.

H.1 Temporal Data for the CRM example based on Physical Annotations in CRM1.psd: CRM1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<CRMdata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:rs="http://www.cs.arizona.edu/tau/tauXSchema/RXSchema"
         xsi:noNamespaceSchemaLocation="repCRM1.xsd"
         xsi:schemaLocation="http://www.cs.arizona.edu/tau/RXSchema RXSchema.xsd">
  <customer>
    <timeVaryingAttribute name="supportLevel" value="gold"
                          vtBegin="2001-02-15" vtEnd="2002-02-15"/>
    <timeVaryingAttribute name="supportLevel" value="platinum"
                          vtBegin="2002-02-15" vtEnd="forever"/>
    <contactInfo>
      <name>Tom</name>
    </contactInfo>
    <directedPromotion>...</directedPromotion>
    <supportIncident>
      <rs:timestamp vtBegin="2001-03-12" vtEnd="2001-04-05"/>
      <product>...</product>
      <description>...</description>
      <action>
        <rs:timestamp vtBegin="2001-03-12" vtEnd="2001-03-20"/>
      </action>
      <action>
        <rs:timestamp vtBegin="2001-03-20" vtEnd="2001-04-05"/>
      </action>
      <resolution>...</resolution>
    </supportIncident>
  </customer>

  <customer>
    <timeVaryingAttribute name="supportLevel" value="gold"
                          vtBegin="2001-01-05" vtEnd="forever"/>
    <contactInfo>
      <name>Bill</name>
    </contactInfo>
    <directedPromotion>...</directedPromotion>
    <supportIncident>
      <rs:timestamp vtBegin="2001-04-02" vtEnd="2001-04-10"/>
      <product>...</product>
      <description>...</description>
      <action>
        <rs:timestamp vtBegin="2001-04-02" vtEnd="2001-04-05"/>
      </action>
      <action>
        <rs:timestamp vtBegin="2001-04-05" vtEnd="2001-04-10"/>
      </action>
      <resolution>...</resolution>
    </supportIncident>
    <supportIncident>
      <rs:timestamp vtBegin="2002-09-12" vtEnd="2002-09-14"/>
      <product>...</product>
      <description>...</description>
      <action>
        <rs:timestamp vtBegin="2002-09-12" vtEnd="2002-09-14"/>
      </action>
      <resolution>...</resolution>
    </supportIncident>
  </customer>
</CRMdata>
```

H.2 Temporal Data for the CRM example based on Physical Annotations in CRM2.psd: CRM2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<valueVaryingRoot xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:rs="http://www.cs.arizona.edu/tau/tauXSchema/RXSchema"
         xsi:noNamespaceSchemaLocation="repCRM2.xsd">
  <CRMdata>
    <rs:timestamp vtBegin="2001-01-05" vtEnd="2001-02-15"/>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Bill</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
    </customer>
  </CRMdata>

  <CRMdata>
    <rs:timestamp vtBegin="2001-02-15" vtEnd="2001-03-12"/>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Tom</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
    </customer>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Bill</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
    </customer>
  </CRMdata>

  <CRMdata>
    <rs:timestamp vtBegin="2001-03-12" vtEnd="2001-03-20"/>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Tom</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
      <supportIncident>
        <product>product1</product>
        <description>...</description>
        <action>action1</action>
        <resolution>...</resolution>
      </supportIncident>
    </customer>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Bill</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
    </customer>
  </CRMdata>

  <CRMdata>
    <timestamp vtBegin="2001-03-20" vtEnd="2001-04-02"/>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Tom</name>
      </contactInfo>
      <supportIncident>
        <product>product1</product>
        <description>...</description>
        <action>action2</action>
        <resolution>...</resolution>
      </supportIncident>
    </customer>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Bill</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
    </customer>
  </CRMdata>

  <CRMdata>
    <rs:timestamp vtBegin="2001-04-02" vtEnd="2001-04-05"/>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Tom</name>
      </contactInfo>
      <supportIncident>
        <product>product1</product>
        <description>...</description>
        <action>action2</action>
        <resolution>...</resolution>
      </supportIncident>
    </customer>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Bill</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
      <supportIncident>
        <product>product2</product>
        <description>...</description>
        <action>action3</action>
        <resolution>...</resolution>
      </supportIncident>
    </customer>
  </CRMdata>

  <CRMdata>
    <rs:timestamp vtBegin="2001-04-05" vtEnd="2001-04-10"/>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Tom</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
    </customer>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Bill</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
      <supportIncident>
        <product>product2</product>
        <description>...</description>
        <action>action4</action>
        <resolution>...</resolution>
      </supportIncident>
    </customer>
  </CRMdata>

  <CRMdata>
    <rs:timestamp vtBegin="2001-04-10" vtEnd="2002-02-15"/>
    <!-- The same as CRMdata from "2001-01-15" to "2001-03-12" -->
  </CRMdata>

  <CRMdata>
    <rs:timestamp vtBegin="2002-02-15" vtEnd="2002-09-12"/>
    <customer supportLevel="platinum">
      <contactInfo>
        <name>Tom</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
    </customer>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Bill</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
    </customer>
  </CRMdata>

  <CRMdata>
    <rs:timestamp vtBegin="2002-09-12" vtEnd="2002-09-14"/>
    <customer supportLevel="platinum">
      <contactInfo>
        <name>Tom</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
    </customer>
    <customer supportLevel="gold">
      <contactInfo>
        <name>Bill</name>
      </contactInfo>
      <directedPromotion>...</directedPromotion>
      <supportIncident>
        <product>product3</product>
        <description>...</description>
        <action>action5</action>
        <resolution>...</resolution>
      </supportIncident>
    </customer>
  </CRMdata>

  <CRMdata>
    <rs:timestamp vtBegin="2002-09-14" vtEnd="forever"/>
    <!-- The same as CRMdata from "2002-02-15" to "2002-09-12" -->
  </CRMdata>
</valueVaryingRoot>
```

I Timestamp Schema Generated for Copy-Based Per-Expression Slicing: tCRM.xsd

The timestamp schema is generated by the stratum for copy-based per-expression slicing. In this schema, all the elements and attributes are timestamped so that the mapping of type related expression is simple.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.cs.arizona.edu/stratum/TCRM"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:rs="http://www.cs.arizona.edu/tau/RXSchema"
           xmlns:tvv="http://www.cs.arizona.edu/tau/Tvv"
           elementFormDefault="qualified" attributeFormDefault="unqualified">

  <xs:import namespace="http://www.cs.arizona.edu/tau/RXSchema"
             schemaLocation="tau/RXSchema.xsd"/>
  <xs:import namespace="http://www.cs.arizona.edu/tau/Tvv"
             schemaLocation="tau/Tvv.xsd"/>

  <xs:element name="CRMdata">
    <xs:complexType mixed="false">
```

```
      <xs:sequence>
        <xs:element ref="rs:timestamp"/>
        <xs:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="customer">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element ref="rs:timestamp"/>
        <xs:element ref="rs:timeVaryingAttribute" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="contactInfo" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="directedPromotion" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="supportIncident" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!-- Definition of contactInfo includes the subelement rs:timestamp -->
  <!-- Definition of directedPromotion includes the subelement rs:timestamp -->

  <xs:element name="supportIncident">
    <xs:complexType mixed="false">
      <xs:sequence>
        <xs:element ref="rs:timestamp"/>
        <xs:element name="product" type="tvv:timeVaryingValueType"/>
        <xs:element name="description" type="tvv:timeVaryingValueType"/>
        <xs:element ref="action" minOccurs="0" maxOccurs="unbounded/>
        <xs:element ref="resolution"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!-- Definition of action includes the subelement rs:timestamp -->
  <!-- Definition of resolution includes the subelement rs:timestamp -->
</xs:schema>
```