Supporting the Procedural Component of Query Languages over Time-Varying Data

by

Dengfeng Gao

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2 0 0 9

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Dengfeng Gao
entitled Supporting the Procedural Component of Query Languages over Time-Varying Data
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Date: April 29 2009

Richard T. Snodgrass

Date: April 29 2009

Bongki Moon

Date: April 29 2009

Christian Collberg

Final approval and acceptance of this dissertation is contingent upon the candidates submission of the final copies of the dissertation to the Graduate College.
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Date: April 29 2009

Dissertation Director: Richard T. Snodgrass

Statement by Author

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: Dengfeng Gao

Acknowledgments

This work would not be possible without the support and encouragement of my advisor, Rick Snodgrass. I chose this topic and began the thesis under his supervision. During my long journey of working on this thesis, he has always been helpful, inspiring, and encouraging.

I would also like to thank the committee who have provided valuable insights and comments at early stage of my dissertation. All faculty that had ever taught me equipped me with the knowledge and skill to study and research on this dissertation.

Many thanks to the department coordinator, Rhonda Leiva, and her staff who have done everything to help me on the various documents and administration requirements.

Dedication

I dedicate this thesis to my parents, my husband, and my children for their endless love and support.

Table of Contents

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

Table of Contents—*Continued*

List of Figures

List of Figures—*Continued*

List of Tables

Abstract

As everything in the real world changes over time, the ability to model this temporal dimension of the real world is essential to many computer applications. Almost every database application involves the management of temporal data. This applies not only to relational data but also to any data that models the real world including XML data. Expressing queries on time-varying (relational or XML) data by using standard query language (SQL or XQuery) is more difficult than writing queries on nontemporal data. In this dissertation, we present minimal valid-time extensions to XQuery and SQL/PSM, focusing on the procedural aspect of the two query languages and efficient evaluation of sequenced queries. For XQuery, we add valid time support to it by minimally extending the syntax and semantics of XQuery. We adopt a stratum approach which maps a τXQuery query to a conventional XQuery. The first part of the dissertation 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 five 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. We also extend a conventional XML query benchmark to effect a temporal XML query benchmark. Experiments on this benchmark show that in-place slicing is the best. We then apply the approaches used in τXQuery to temporal SQL/PSM. The stratum architecture and most of the time-slicing techniques work for temporal SQL/PSM. Empirical comparison is performed by running a variety of temporal queries.

Chapter 1
Introduction

Time is an important aspect of all real-world phenomena. Events occur at specific points in time; entities and the relationships among entities exist over time. The ability to model this temporal dimension of the real world and to respond to changes in the real world is essential to many computer applications, such as accounting, inventory control, law, medical records, reservation systems, scientific data analysis.

Almost every database application involves the management of temporal data. This aspect has been acknowledged and long studied in the field of temporal databases [51]. It truly applies not only to relational data but also to any data that models the real world. Recently, Extensible Markup Language (XML) [8] has become the emerging standard for data representation and exchange on the web. There is a large amount of XML data being created and stored [38]. Similarly, XML data changes over time with the creation, modification, and deletion of the XML documents. The management of dynamic XML documents requires the adoption of temporally enhanced data models and systems.

Querying data is an important aspect in relational databases as well as in XML data management. The standard query language for relational databases is SQL, which is supported by most commercial relational database management systems. More recently, most DBMSs provide more powerful SQL language constructs that support computational completeness. These language constructs are also part of the SQL standard (control statements and persistent stored modules (PSM)) [49]. We consider the control statements and PSM as the procedural components of SQL. Although each commercial DBMS has its own idiosyncratic syntax and semantics, stored routines are widely used in database applications. The reasons are as follows. PSM

provides the ability to compile and optimize SQL statements and the corresponding database operations once and then execute them many times on demand. This represents a significant savings in resource utilization and savings in the time required to execute those statements. The computational completeness of the language enables complex calculations. Stored SQL facilities allow users to share common functionality and encourage code reuse, thus saving development time, money, and other resources [37].

It has been shown that queries on temporal data are often hard to express in conventional SQL [47] and the average temporal query/modification is three times longer than the nontemporal equivalent [45]. This led to a large number of research works on temporal query languages. Previous research [47] successfully extended SQL to add temporal support while guaranteed the new temporal query language is compatible to the conventional SQL. However, the temporal semantics of the procedural components has never been defined. Temporal applications may potentially benefit from temporal query languages because queries and modifications for these applications are simplified and easier to maintain. To benefit temporal database applications more generally, a temporal SQL language should include stored modules.

In the XML world, the counterpart of SQL is XQuery [5]. 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, Oracle [17], IBM [28], and Microsoft [16], have all released early implementations of XQuery. XQuery is not only a query language, but also a programming language with computational completeness. It supports user-defined functions, similar to PSM in SQL. Our research [25] shows that expressing queries on temporal data is harder than writing queries on nontemporal data in XQuery. Compared to temporal SQL, there has been even less research work on temporal XQuery, since the problem of processing non-temporal XML queries efficiently is itself a hot topic. Most of the existing research work on temporal XML queries focuses on the versioning of XML documents

[13, 14, 35]. Some have proposed transaction-time extensions to XPath (a part of XQuery) [20]. Others proposed temporal extensions in multiple time dimensions to FLWR expression (also a part of XQuery) [27]. These extensions do not guarantee the easy migration of legacy applications to temporal systems.

In the context of databases, two time dimensions are of general interest: valid time and transaction time [46]. *Valid time* denotes the time a fact was true in reality. *Transaction time* is the time during which the fact was recorded in the database. These two dimensions are orthogonal. A data model supporting neither is termed a *snapshot*, as it captures only a single snapshot in time of both the database and the real world modeled by the database. A data model that supports both valid time and transaction time is termed *bitemporal* [29].

This dissertation presents minimal valid-time extensions to XQuery and SQL/PSM, focusing on the procedural aspect of the two query languages and efficient evaluation of sequenced queries. Such languages make it easy to migrate legacy database applications to work with temporal databases.

This dissertation comprises nine chapters besides this introduction. They can be divided into four major parts. The first part is Chapter 2 which is a survey of related work. The second part consists of Chapters 3 to 6. This part focuses on temporal XQuery. Chapter 3 briefly introduces the temporal XML data model and then proposes the temporal XML query language τXQuery. Both the syntax and the semantics of τXQuery are provided in this chapter. Chapter 4 explains the advantages of using a stratum approach to implement the τXQuery query processor and illustrates the architecture of the stratum. We propose several timeslicing techniques in Chapter 5 to optimize the translation of τXQuery queries to semantically equivalent XQuery queries. The performance of different timeslicing techniques is compared empirically in Chapter 6. The third part of the dissertation discusses how to apply the approaches used in τXQuery to temporal SQL/PSM. Chapter 7 describes the problem of temporal SQL/PSM and our proposals of the syntax and semantics of

temporal PSM. In Chapter 8, we observed in temporal SQL/PSM the counterparts of some of the techniques used for implementing τXQuery. The performance of these techniques are evaluated in Chapter 9. Finally, the dissertation is concluded in Chapter 10 which summarizes the work and its major contributions and holds an outlook to future work.

Chapter 2
Related Work

In this chapter, we revisit the research on temporal query languages for relational databases and temporal XML. Since query languages are based on specific data models, we introduce the related temporal data models as well.

2.1 Temporal Relational Data Models

There have been a large number of temporal data models for relational databases proposed in the past two decades [39]. Some models are defined only over valid time or transaction time; others are defined over both. A temporal data model should simultaneously satisfy many goals. The experience of the past twenty years and dozens of data models appearing in the literature [39] demonstrate that such an ideal temporal data model does not exist. A conceptual temporal data model uses a suite of data models to achieve goals that no single data model can. This is the data model proposed as the foundation of TSQL2 [30]. This language employs the Bitemporal Conceptual Data Model as its underlying data model, which retains the simplicity and generality of the relational model. A separate, *representational* data model of equivalent expressive power, employed for implementation, ensures high performance. Other, *presentational* data models may be used to render time-varying behavior to the user or application.

A natural and frequently used way to represent a bitemporal relation is a 1NF representation proposed by Snodgrass [44], which allows the use of existing, well-understood implementation techniques. This representation scheme associates each tuple in a conventional relation with valid timestamps and transaction timestamps.

Let the bitemporal relation schema have the non-temporal attributes A_1, ..., A_n. It is represented by a snapshot relation schema R as follows.

$$R = (A_1, ..., A_n, T_s, T_e, V_s, V_e)$$

The additional attributes T_s, T_e, V_s, V_e are atomic valued timestamp attributes containing a starting and ending transaction-time chronon and a starting and ending valid-time chronon, respectively. A *chronon* is the smallest time unit [29]. For example, consider a relation recording customer information, which includes a customer's name and his/her support level. Support level of a customer indicates how valuable the customer is to the company. We assume that the granularity of chronons is one day for both valid time and transaction time. Figure 2.1 shows the 1NF representation of the temporal relation.

CName	SupportLevel	T_s	T_e	V_s	V_e
Bill	Gold	6/5	6/9	6/10	6/15
Bill	Gold	6/10	6/14	6/5	6/15
Bill	Gold	6/15	UC	6/5	6/9
Bill	Platinum	6/15	UC	6/10	6/15
Jane	Gold	6/20	UC	6/25	6/30

FIGURE 2.1. 1NF representation of a temporal relation

Customer Bill was a gold customer for the period from June 10th to June 15th, and this fact is recorded in the database predictively on June 5th. On June 10th, the customer service department discovers an error. Bill was actually a gold customer since June 5th. The database is corrected on June 10th. On June 15th, the customer service department finds that Bill was actually promoted to a platinum customer since June 10th, and the database is corrected the same day. After that, the information about Bill remains unchanged, which is represented by UC (*until_changed*). Another customer Jane is a gold customer for the period from June 25th to June 30th, and this is recorded in the database on June 20th.

There are many other temporal data models, each of which may be appropriate under some circumstances. However, those data models are not related to our research on temporal SQL/PSM since our work is based on the temporal query language SQL/Temporal, which uses the data model presented above.

2.2 Temporal Query Languages for Relational Data

Many temporal query languages have been proposed. These languages are based on different conventional query languages varying from relational algebra, tuple calculus-based query language, to domain calculus-based query language. Since SQL is the standard query language for relational databases, the SQL-based temporal query languages have more potential of being used.

SQL/Temporal [47] is a query language obtained by adding valid-time support to SQL3. It was proposed as part of the new SQL standard. The classification of temporal queries to current query, sequenced query, and representational query was introduced in this language to ensure *upward compatibility* and *temporal upward compatibility* [4, 6]. Upward compatibility guarantees that the existing applications running on top of the temporal system will behave exactly the same as when they run on the legacy system. Temporal upward compatibility ensures that when an existing database is transformed into temporal database, the legacy queries apply to the current state of the database. These two properties guarantee the easy migration of a legacy application to a temporal system. We use this classification in the language design of τXQuery. SQL/Temporal only defines the temporal semantics of data definition statements and data manipulation statements. It does not support temporal persistent stored modules (PSM). Our work on temporal SQL/PSM is an integration of SQL/Temporal and persistent stored modules.

2.3 Implementation of Relational Temporal Query Languages

Implementing a DBMS with built-in temporal support from scratch is a daunting task that may only be accomplished by major DBMS vendors that already have a DBMS to modify and have large resources available. Torp et al. proposed the layered strategy to implement temporal DBMS [52]. A layer is inserted between the conventional DBMS and the users of the temporal DBMS. The main task of this layer is to map the queries written in temporal SQL to semantically equivalent queries in conventional SQL. The intension was to maximally reuse the facilities of an existing SQL implementation. We adopted this strategy for a similar reason. Slivinskas et al. explored the query optimization issues for temporal DBMS implemented by a middle-ware approach [42, 43]. This dissertation does not address the optimization of temporal query languages.

2.4 Temporal XML Data Models

In recent years, a crop of research work addressed temporal and versioning aspects in the Web and, in particular, in the management of XML documents.

In terms of data model, there are three different categories. Approaches that fall in the first category focus on the representation of changes, where different versions of data are produced by updates. In these approaches, temporal attributes are often used to timestamp stored versions [2, 12, 13, 53]. They represent the time the updates occurred and thus, have the (implicit) semantics of transaction time. The second category includes the approaches considering the classical notion of valid time [10, 26, 54] or valid and transaction time [21, 34]. For example, the "Valid Web" approach [26] is an infrastructure designed to represent temporal Web documents with timestamps explicitly encoded by the document authors to assign validity to information contents. An example that considers both valid time and transaction time is τXSchema. Similar to TSQL2, τXSchema uses a suite of schemas to separate the logical schema from

the physical representation. This data model serves as the base of our research on temporal XQuery. We give more details of τXSchema in Section 3.2. The last category considers more time dimensions than valid time and transaction time. Data models in this category usually focus on a particular application. For example, Grandi et al. designed a data model to manage normative text in XML format [27]. They identified two more time dimensions: *publication time* (the time of publication of the norm) and *efficacy time* (the time the norm can be applied to a concrete case). These time dimensions may not be applicable to general XML applications.

2.5 Temporal XML Query Languages

Concerning temporal XML query languages, there has been some work addressing the transaction time dimension of XML [12, 13, 15, 36]. These papers focus on XML versioning, including detecting and querying the changes in XML documents. Dyreson et al. proposed a framework for querying meta-data properties including temporal information in semistructured data [21]. This work can be viewed as an extension to a conventional semistructured database. Grandi and Mandreoli [26] introduced a valid time extension to XQL to express temporal predicates. In our terminology, their approach would be considered to support representational queries with additional predicates. Similarly, Wang and Zaniolo [54, 55, 56] added temporal predicates to XQuery to query histories of XML-published relational databases. The normative text data model proposed by Grandi et al. [27] supports temporal queries following a particular XQuery format, specifically a simplified FLWR expression, in which temporal predicates can be specified. A temporal extension to XPath was proposed by Dyreson [20] in which a transaction time axis is added to path expression. Later, Zhang and Dyreson added a valid time axis to XPath [58]. Our work is different in that we added the valid time extension to the entire language of XQuery. We also guarantee the temporal upward compatibility of the temporal query language, which

is not addressed by other temporal XML query languages.

2.6 Implementation of Temporal XML Query Languages

Two sets of authors proposed the implementation of Temporal XML Query. Grandi et al. implemented a prototype system to manage temporal normative documents by means of a stratum [27]. The temporal XML documents are stored as a CLOB in a relational DBMS that supports XML/SQL query. The stratum accepts query expressions which can involve both temporal constraints and search keywords and then maps each request to a semantically equivalent XML/SQL expression to be passed to the relational DBMS. The result query is a `SELECT` statement with path expression specifying the target nodes. The temporal constraints are translated to predicates in a `WHERE` clause. Wang and Zaniolo's approach is similar [55]. Their system translates XQuery expressions with temporal constraints to SQL statements to be submitted to a relational DBMS that stores the data. The details of how the translation is done is not provided. In this dissertation, we discuss how to translate expressions in temporal XQuery language to conventional XQuery expressions.

Chapter 3
Temporal XQuery

In this chapter, 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 to make sure τXQuery is both upward compatible and temporal upward compatible to XQuery. Section 3.1 gives an example that illustrates the benefit of temporal support within the XQuery language. Temporal XML schema is briefly introduced in Section 3.2. Section 3.3 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 3.5. The example τXQuery queries are evaluated on an XQuery engine, Galax, and their results are shown in Section 3.6. In the following two chapters, we discuss the details of a stratum to implement τXQuery on top of a system supporting conventional XQuery and the time-slicing techniques used in the stratum.

3.1 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 [3, 24]. An important task is to collect and

analyze historical information on customer interactions. As Ahlert emphasizes, "It is necessary for an organization to develop a common strategy for the management and use of all customer information" [1], termed *enterprise customer management*. This requires communicating information on past interactions (whether by phone, email, or web) to those who interact directly with the customer (the "front desk") and those who analyze these interactions (the "back desk") for product development, direct marketing campaigns, incentive program design, and refining the web interface. Given the disparate software applications and databases used by the different departments in the company, using XML to pass this important information around is an obvious choice.

Figure 3.1 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

```
<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 3.1. A CRM XML document

a means of adding time to an XML schema to realize a *representational schema*, which is itself a correct XSchema [23], 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 3.2. In this particular temporal XML document, a time-varying attribute is *represented* as a `timeVaryingAttribute` element, and that a time-varying element is represented with one or more *versions*, each containing one or more `timestamp` sub-elements. The valid-time period is represented with the beginning and ending instants, in a closed-open representation. Hence, the "gold" attribute value is valid for the day September 19 through the day March 19; March 19 is not included. (Apparently, a support level applies for six months.) Also, the valid period of an ancestor element (e.g., `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.

```
<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="2004-9-19"/>
     ...
     <supportIncident>
       <timestamp vtBegin="2002-2-11" vtEnd="2002-2-29"/>
       ...
       <action>
         <timestamp vtBegin="2002-2-11" vtEnd="2002-2-21"/>
         <who> ... </who>
         ...
       </action>
       <action> <timestamp .../> ... </action>
     </supportIncident>
  </customer>
</CRMdata>
```

FIGURE 3.2. A time-varying CRM XML document

Consider now an XQuery query on the static instance in Figure 3.1, "What is the average number of open support incidents for gold customers?" This is easily expressed in XQuery as

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

This query binds the variable `$c` to each `customer` element whose support level is gold, counts the number of support incidents of each customer, and computes the average number of support incidents for these customers.

Now, if the analyst wants the history of the average number of open support incidents for 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.) The temporal query written in XQuery is as follows.

```
let $timepoints = distinct-values(
    for $p in (document("CRM.xml")//timestamp union
               document("CRM.xml")//timeVaryingAttribute)
    for $t in ($p/@vtBegin, $p/@vtEnd)
    order by $t return $t)
for $index := 1 to count($timepoints)-1
let $begin := item-at($timepoints, $index)
let $end := item-at($timepoints, $index+1) return
    <timeVaryingValue>
      <timestamp vtBegin=$begin vtEnd=$end />
      <value>
      avg(for $c in document("CRM.xml")//customer
```

```
        where some $sl in $c/timeVaryingAttribute satisfies
                            ($sl/@vtBegin <= $begin and
                             $sl/@vtEnd > $begin and
                             $sl/@name = "supportLevel" and
                             $sl/@value = "gold") return
         count(for $si in $c/supportIncident
               where some $ts in $si/timestamp satisfies
                                  ($ts/@vtBegin <= $begin and
                                   $ts/@vtEnd > $begin)
               return $si)
    </value>
</timeVaryingValue>
```

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.

3.2 Temporal XML Schema

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

The temporal XSchema model, τXSchema [18] 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 an 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 3.2) 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 Chapter 5.

3.3 τXQuery

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.

3.3.1 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 for (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 3.2, 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.

3.3.2 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 for 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 3.2 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.

3.3.3 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, for 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 3.2 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.

3.3.4 Compatibility

Representational queries have the same semantics (and syntax!) as XQuery. This indicates τXQuery is a superset of XQuery, which ensures τXQuery is *upward compatible* 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 [4] 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 are specified in the query to check if the namespace of the representational schema (e.g., `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.

3.4 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 [50]. The reason we use denotational semantics

is that we would like to demonstrate the mapping formally. By writing the denotational semantics for each language construct, we accurately show that the whole language can be mapped. Alternative approaches to show the mapping include using English to describe the mapping, using graph to illustrate the mapping, or using pseudo codes to show the mapping algorithms. None of them are as accurate as denotational semantics.

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 Chapter 5. 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 working draft [19] 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 ⟨Q⟩ is identical to that of XQuery.

⟨TQ⟩ ::= (`current` | `validtime` (`[`⟨BT⟩`,` ⟨ET⟩`]`)$^?$ | `rep validtime`)$^?$ ⟨Q⟩

The semantics of ⟨TQ⟩ 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 3.3.4, 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 $cur\,[\![\langle \text{TQ} \rangle]\!]$

3.4.1 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` (see Appendix A) contains definitions of the `timestamp` and `timeVaryingAttribute` elements. The other namespace `tau` is defined for the semantic mapping. All the auxiliary functions and variables used for the mapping have this prefix. We use a new semantic function *snapshot* ⟦ ⟧ *w*hich takes an additional parameter, an XQuery expression that evaluates to the `xs:dateTime` type. As with

other semantic functions utilized here, the domain is a τXQuery expression (a string) and the range is an XQuery expression (also a string).

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

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

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

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.

3.4.2 Representational Queries

The mapping for representational queries is trivial.

rep ⟦`rep validtime` ⟨Q⟩⟧ = *rep* ⟦⟨Q⟩⟧
cur ⟦`rep validtime` ⟨Q⟩⟧ = *rep* ⟦⟨Q⟩⟧
rep ⟦⟨Q⟩⟧ = ⟨Q⟩

The only thing needed is to remove the reserved word `rep validtime`. This mapping obviously ensures that τXQuery is upward compatible with XQuery.

3.4.3 Sequenced Queries

In a sequenced query, the reserved word `validtime` is followed by an optional time 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. For

example, the query asking for the history of the average number of open support incidents for gold customers during year 2001 would be expressed as follows.

```
validtime [2001-01-01, 2002-01-01]
avg(for $c in document("CRM.xml")//customer[@supportLevel="gold"]
    return count($c/supportIncident))
```

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` ⟨Q⟩⟧ = *cur* ⟦`validtime` ⟨Q⟩⟧ =
seq ⟦⟨Q⟩⟧ `$tau:period("1000-01-01", "9999-12-31")`

When there is no valid-time period specified in the query, the query is evaluated in the whole timeline the system can represent. 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 [`⟨BT⟩`,`⟨ET⟩`]` ⟨Q⟩⟧ = *cur* ⟦`validtime [`⟨BT⟩`,`⟨ET⟩`]` ⟨Q⟩⟧ =
seq ⟦⟨Q⟩⟧ `tau:period(`⟨BT⟩`,` ⟨ET⟩`)`

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

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 straightforward approach to map a sequenced query to XQuery, based on the following simple observation first made when the semantics of temporal aggregates were defined [48]: 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.

⟨Q⟩ ::= ⟨QueryProlog⟩ ⟨QueryBody⟩

seq ⟦⟨Q⟩⟧ p =

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

The namespace `tvv` defines the sequenced time-varying value type needed in the mapping. The schema that defines `tvv` is given 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.

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 3.2.

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 Chapter 5, we examine more sophisticated slicing strategies). Since the documents appearing in both ⟨QueryProlog⟩ and ⟨QueryBody⟩ need to be time-sliced, we define *seq* ⟦⟨QueryProlog⟩⟧ p and *timeslice* ⟦⟨QueryBody⟩⟧ t further. In ⟨QueryProlog⟩, only the function definitions need to be mapped. We add an additional parameter (a time point) to each user-defined function and use this time point to slice the document specified in the function.

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

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

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

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

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

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

The function `document()` 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 5.2.

⟨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) as xs:node
{  document("term.xml")//term[name = $s]  }
```

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

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 `timeVaryingAttribute` 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 `order by` 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 an `order by` 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.

3.4.4 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 3.3.4, 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* [29, 44]. 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.

3.5 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 3.4.3 does not work. However, in Section 5.2, 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 [44].

Theorem 2. *$\tau XQuery$ 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. □

3.6 Example Queries and Results

The three example queries mentioned in Section 3.3 have been mapped to XQuery and tested on an XQuery engine Galax [31]. Since Galax does not support all the features of XQuery, we made a few changes to the auxiliary functions to enable the test. For example, it does not support the data type `dateTime`. We just used `string` as a substitution. As mentioned in Section 3.2, 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.

3.6.1 Current Query

Query:
What is the average number of open support incidents for gold customers?
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 $c in tau:snapshot(document("CRM1.xml"), current-dateTime())//
                                          customer[@supportLevel="gold"]
    return count($c/supportIncident))
```

Result: `0`

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

3.6.2 Sequenced Query

The sequenced query is mapped using the slicing approach described in Section 3.4.3. The query results from the two instances are exactly the same. None of them are coalesced.

Query:
What is the history of the average number of open support incidents for gold customers?
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>
```

3.6.3 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, for gold customers, 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`

Now, we have seen the syntax and semantics of the temporal XML query language τXQuery and the example τXQuery queries. Next chapter discusses the implementation of this language using a stratum approach.

Chapter 4
The Stratum Approach

In Section 3.4.4, we discussed the pleasing symmetry in the formal semantics of the three modes. 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 [52]. 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 4.1. The dashed rectangle indicates the boundary of the stratum. When a query is input, the initial keyword is examined and the 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 3.4. 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*$[\![\ ]\!]$ semantic function discussed briefly in Section 3.4.3 is implemented by the sequenced query mapper. The documents mentioned in the query (and in functions called directly or indirectly by the query) can be determined from a syntactic analysis of the query; 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

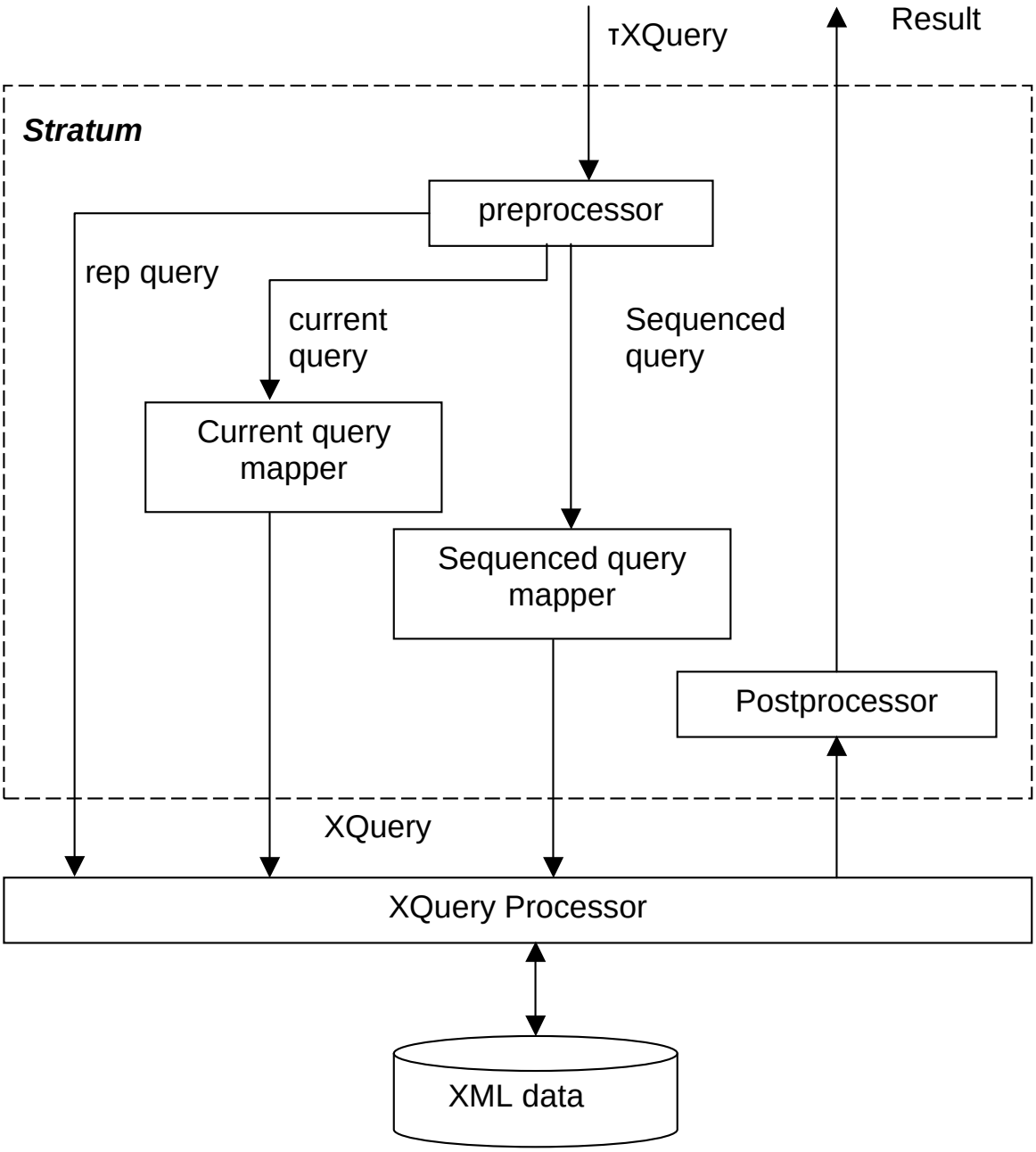


FIGURE 4.1. Architecture of the τXQuery stratum

coalesces the query results. *Coalescing* in relational temporal databases is a unary operator [33, 29]; 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 3.3, 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 3.3.2. It is possible that the average is 5 during two separate but adjacent periods. In this case, the result is uncoalesced (the result is represented by two elements when one would do). Coalescing this result will merge the two elements into one.

Coalescing temporal XML data is different in many aspects from coalescing relational data. It is an open question whether coalescing can be done efficiently in XQuery, or whether this computation is best done in the stratum. It is an interesting topic for future research which we discuss in Chapter 10.

It is obvious that processing a representational query is trivial to the stratum since no mapping is needed. The most challenging part is to map a sequenced query. We focus on the techniques for mapping sequenced queries in next chapter.

Chapter 5
Optimization of Slicing

In Section 3.4.3, 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 `timeVaryingAttribute` element contained in each document. We call this method *maximally-fragmented time-slicing*. (We emphasize that this approach is far more efficient than taking a timestamp of the document at every time point in which it is valid, termed *unfolding* in the context of temporal relations [32]. 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 3.2, 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 3.3.2 does not go all the way down to `action`. In particular, examining Figure 3.2 indicates that a constant period of [2002-4-11–2002-4-29) is sufficient, without being broken into two periods at 2002-4-21. Figure 5.1 illustrates this situation using the timevarying information of an example customer. The support level of the customer changed at some point (indicated by the dot on the segment of support level). The customer had two support incidents, each of which had two actions. The constant periods computed by maximally-fragmented slicing are shown as the segments in the lowest line. The circled point is one of the extra slices since this changing point of `action` does not impact the query results. 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 chapter, 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.

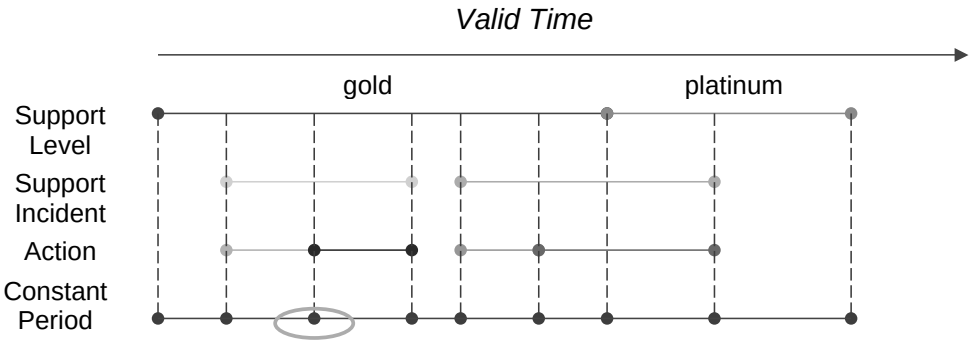


FIGURE 5.1. Extra Slicing in Maximally-Fragmented Slicing

5.1 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 by selected node slicing.

sn ⟦⟨Q⟩⟧ *p* =

```
import schema namespace rs = "http://www.cs.arizona.edu/tau/RXSchema"
       at "RXSchema.xsd"
import schema namespace tvv = "http://www.cs.arizona.edu/tau/Tvv"
       at "TimeVaryingValue.xsd"
declare namespace tau = "www.cs.arizona.edu/tau/Func"
```
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 Section 3.3.2, 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`. This can be observed by comparing the number of slices in Figure 5.2 with that in Figure 5.1.

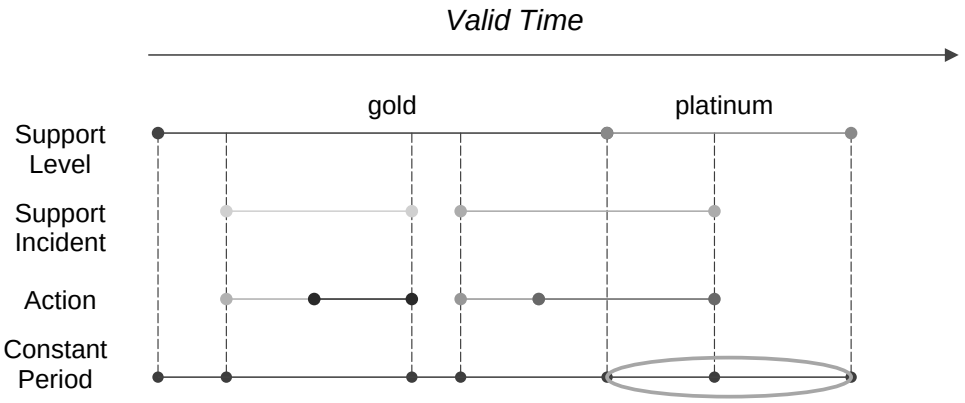


FIGURE 5.2. Selected Node Slicing

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 [41], one in 23 queries in XOO7 [9], one in 16 queries in XBench [57], and four in eight queries in XMach-1 [7].

This method and the maximally-fragmented time-slicing method both 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. In Figure 5.2, the circled point is one of the extra slices. The support level of the customer at this point was "platinum", thus the support incident at this point does not contribute to the result. Time-slicing this untouched part is wasted work. In the next two sections, we present methods that avoid slicing the unused subtrees.

5.2 Per-Expression Slicing

XQuery is a functional language which allows various kinds of expressions to be nested with full generality. The syntax tree of the query used in Section 3.3.2 is shown in Figure 5.3. The query is a function call, which has a for expression embedded as a parameter. The for expression further has a path expression and another function

call embedded. The evaluation of the expression is in a bottom up fashion. The path expression in the lower left part of the syntax tree should be evaluated first. The result is a sequence of `customer` elements, each of which is a subtree of the input XML document, Then each `customer` element is bound to the variable `$c` and the lower right subexpression is evaluated.

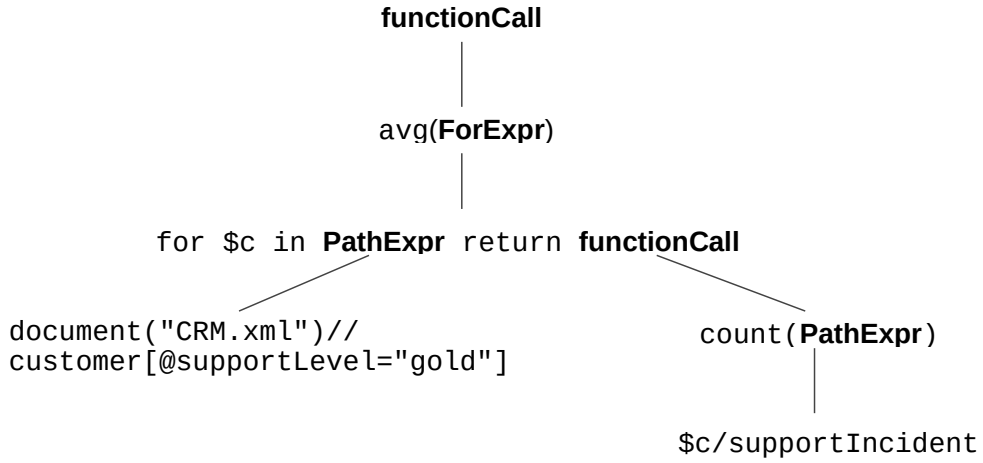


FIGURE 5.3. The Syntax Tree

Per-expression slicing maps each sequenced subexpression individually. It slices the subtree (data) 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 (data). The sequenced version of the current subexpression then is evaluated on the time-sliced subtree (data). The result, a sequence of trees (data) 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 (usually a higher level in the syntax tree). The constant periods in the subsequent level are shorter than, and contained within, the constant periods in the previous level. Thus, those unused subtrees (data) are pruned before they are time-sliced. Consider the sequenced query in Figure 5.3 evaluated on the example data shown in Figure 5.1. The result of the lower left path expression is the `customer` elements with only the

valid descendants. Figure 5.4 illustrates the valid parts of the `customer` element. The evaluation of any subsequent expression will not consider the invalid parts of this customer.

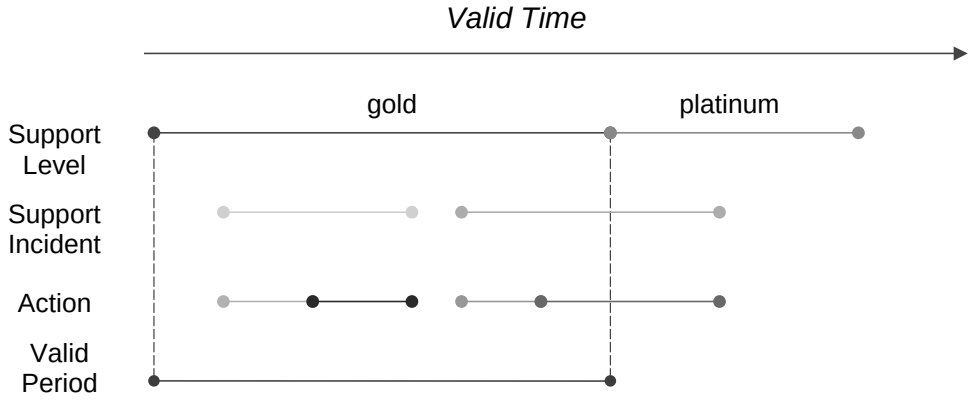


FIGURE 5.4. Intermediate Result

Since some of the nodes do not have timestamps in the original temporal XML document, we need a way to remember the valid period for such nodes. In this 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.

5.2.1 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 [19], a subset of the XQuery grammar provided by the W3C, defining the semantics to map the core grammar of τXQuery is sufficient.

The normalized result of the example query mentioned in Section 3.3.2 is shown in Figure 5.5. This result is obtained by applying the normalization formally defined in W3C working draft [19]. The only difference is we change the prefix `fs` to `tau`,

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

FIGURE 5.5. Normalizing the example query

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. ⟨Q⟩ ::= ⟨QueryProlog⟩ ⟨QueryBody⟩

   As with the mapping function defined in previous sections, before ⟨QueryProlog⟩ and ⟨QueryBody⟩ 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. ⟨QueryProlog⟩ ::= (⟨NamespaceDecl⟩
   | ⟨XMLSpaceDecl⟩
   | ⟨DefaultNamespaceDecl⟩
   | ⟨DefaultCollationDecl⟩
   | ⟨SchemaImport⟩)* ⟨FunctionDefn⟩*

   ⟨XMLSpaceDecl⟩ and ⟨DefaultCollationDecl⟩ 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 `timeVarying-Attribute` which could appear as children of `customer` and all its subelements.

The ⟨DefaultNamespaceDecl⟩ 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. ⟨FunctionDefn⟩ ::= `define function` ⟨FuncName⟩ `(`⟨ParamList⟩$^{?}$`)`
   `as` ⟨SequenceType⟩
   {⟨ExprSequence⟩}

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

⟨ParamList⟩ ::= ⟨Param⟩ (, ⟨Param⟩)*

$cb\,[\![\langle \text{ParamList} \rangle]\!]\ p = cb\,[\![\langle \text{Param} \rangle]\!]\ p\ (\ ,\ cb\,[\![\langle \text{Param} \rangle]\!]\ p\ )^*$

For the rest of the section, we will omit such obvious semantic functions that mirror the productions.

⟨Param⟩ ::= ⟨SequenceType⟩ `$`⟨VarName⟩

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

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.

⟨SequenceType⟩ ::= (⟨ItemType⟩⟨OccurenceIndicator⟩) | `empty`

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

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.

⟨ItemType⟩ ::= ((`element` | `attribute`)⟨ElemOrAttrType⟩$^{?}$)
| ⟨AtomicType⟩
| `node`
| `processing-instruction`
| `comment`
| `text`
| `document`
| `item`
| `untyped`
| `atomic value`

⟨AtomicType⟩, `atomic value`, and `untyped` are mapped to `tvv:timeVarying-ValueType`. 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. ⟨QueryBody⟩ ::= ⟨ExprSequence⟩$^{?}$

   ⟨ExprSequence⟩ ::= ⟨Expr⟩ (`,`⟨Expr⟩)*

   ⟨Expr⟩ ::= ⟨ForExpr⟩ | ⟨LetExpr⟩

   ⟨ForExpr⟩ ::= (⟨ForClause⟩ ⟨OrderByClause⟩$^{?}$ `return`)* ⟨TypeswitchExpr⟩
   ⟨ForClause⟩ ::= `for` ⟨SequenceType⟩$^{?}$ `$`⟨VarName⟩ `in` ⟨Expr$_1$⟩
   ⟨OrderByClause⟩ ::= `order by` ⟨Expr$_2$⟩ ⟨OrderModifier⟩

$cb\,[\![\langle\text{ForExpr}\rangle]\!]\ p =$

```
for $tau:i in cb[[⟨Expr1⟩]] p
for $tau:p in tau:periods-of($tau:i)
let (cb[[⟨SequenceType⟩]] p)? $⟨VarName⟩ :=
                tau:copy-restricted-subtree($tau:p, $tau:i)
for $tau:i1 in cb[[⟨Expr2⟩]] $tau:p
for $tau:p1 in tau:periods-of($tau:i1)
order by $tau:i1 ⟨OrderModifier⟩
return cb[[⟨TypeswitchExpr⟩]] $tau:p1
```

The auxiliary function `periods-of()` returns all the timestamps associated with the input node. Since the sequence returned by $\langle\text{Expr}_1\rangle$ could contain multiple versions of an item, valid over different periods of time, the following ⟨OrderByClause⟩ should be evaluated in each of these periods. Similarly, The evaluation of sequenced $\langle\text{Expr}_2\rangle$ produces more and shorter valid periods. ⟨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.

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

   $cb\,[\![\langle\text{LetExpr}\rangle]\!]\ p =$

```
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.

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

   The mapping of ⟨TypeswitchExpr⟩ is similar to that of ⟨LetExpr⟩.

   $cb\,[\![$⟨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
   ```

7. ⟨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 $tau:s := tau:snapshot($tau:b, $tau:p/@vtBegin) return
     if ($tau:s)
     then cb[[⟨Expr2⟩]] $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.

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

   $cb\,[\![\langle\text{ValidateExpr}\rangle]\!]\ p =$

   ```
   let $tau:s := cb [[⟨Expr⟩]] p
   for $tau:p in tau:const-periods(p, $tau:s) return
     validate ( temSC [[⟨SchemaContext⟩]] )?
                tau:copy-restricted-subtree($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 analog. 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
```

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

$cb\,[\![\langle\text{CastExpr}\rangle]\!]\ p =$

```
let $tau:s := cb [[⟨ExprSequence⟩]] p
for $tau:p in tau:const-periods(p, $tau:s)
let $tau:v := cast as ⟨AtomicType⟩ tau:snapshot(
   tau:copy-restricted-subtree($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 atomic type to another atomic type, we cast the snapshot of the expression at each constant period and wrap the cast result in a `timeVaryingValue` element.

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

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⟩?}

$cb\,[\![\langle\text{ComputedAttributeConstructor}\rangle]\!]\ p =$

```
let $tau:s := cb⟦⟨ExprSequence⟩⟧ p
for $tau:p in $tau:const-periods(p, $tau:s) return
  element timeVaryingAttribute
  {
    attribute name {⟨QName⟩},
    attribute value {tau: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 sequence of `timeVaryingAttribute` elements. When the attribute name itself is an expression, the valid periods of the attribute are computed from both the ⟨Expr⟩ and the ⟨ExprSequence⟩.

11. ⟨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:ta in $`⟨VarName⟩`/timeVaryingAttribute[@name=`⟨NameTest⟩`]`
`return tau:copy-restricted-subtree(`p`, $tau:ta)`

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 is needed when the ⟨NodeTest⟩ is a wildcard. It hides all the subelements added by τXQuery from the user.

$cb\,[\![\texttt{\$}\langle\text{VarName}\rangle\texttt{/}\langle\text{ForwardStep}\rangle]\!]\ p =$

```
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.

$cb\,[\![\langle\text{Prefix}\rangle\texttt{:}\langle\text{LocalName}\rangle]\!]\ p = temNode\,[\![\langle\text{Prefix}\rangle]\!]\texttt{:}\langle\text{LocalName}\rangle$

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::`

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

$cb\,[\![\langle\text{Literal}\rangle]\!]\ p =$

```
<timeVaryingValue>
  p,
  <value>⟨Literal⟩</value>
</timeVaryingValue>
```

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

13. ⟨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.

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

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.

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

```
  let $tau:par1 := cb [[⟨Expr1⟩]] p
  let $tau:par2 := cb [[⟨Expr2⟩]] p
  for $tau:p in tau:all-const-periods(p,
                          ($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`

14. $cb\,[\![\$\langle\text{VarName}\rangle]\!]\ p =$

    `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 in Section 3.3.2, the τXQuery processor first normalizes it to the query shown in Figure 5.5. This normalized query is then mapped to the XQuery query in Figure 5.6. 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.

5.2.2 In-Place Per-Expression Slicing

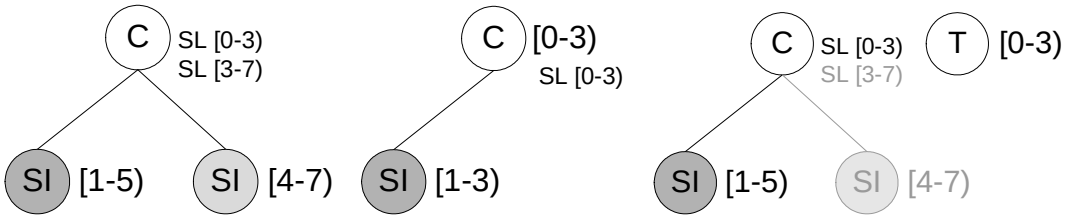
Rather than timestamping all the intermediate results, in-place per-expression slicing keeps all the intermediate results with the document. To record the valid period of these intermediate results, it puts the intermediate results and their actual timestamps in one sequence in the form of (item, timestamp, 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 5.7. Figure 5.7(a) shows a `customer` element in the original document. The customer has two timestamped sub-elements `supportIncident`. The support level of this customer changed from gold to platinum at time point 3. Consider the syntax tree in Figure 5.3. When the path expression in the lower left part

```
{-- validtime avg(for $c in --}
let $tau:par :=
  (for $tau:i in
   {-- let $tau:sequence:=document("CRM.xml") return --}
    (let $tau:s := tau:copy-restricted-subtree(tau:period("1000-01-01",
                                "9999-12-31"), document("CRM.xml"))
     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
         {-- for $tau:dot in $tau:dot/descendant-or-self::customer return --}
         for $tau:i2 in
           (for $tau:step in $tau:dot/descendant-or-self::customer return
              tau:copy-restricted-subtree($tau:p1, $tau:step))
         for $tau:p2 in tau:periods-of($tau:i2)
         let $tau:dot := tau:copy-restricted-subtree($tau:p2, $tau:i2) return
           {-- if expression--}
           for $tau:c in
             (for $tau:ta in $tau:dot/timeVaryingAttribute[@name="supportLevel"]
              return tau:copy-restricted-subtree($tau:p2, $tau:step))
           let $tau:p3 := tau:periods-of($tau:c)
           let $tau:b := element timeVaryingValue
                         {$tau:p3,
                          $tau:c/@value = "gold"}
           let $tau:s := tau:snapshot($tau:b, $tau:p3/@vtBegin) return
             if ($tau:s)
             then tau:copy-restricted-subtree($tau:p3, $tau:dot)
             else ())
   for $tau:p in tau:periods-of($tau:i)
   let $c := tau:copy-restricted-subtree($tau:p, $tau:i) return
     {-- count(for $tau:dot in $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 $tau:dot := tau:copy-restricted-subtree($tau:p1, $tau:s1) return
            {-- $tau:dot/child::supportIncident))) --}
            for $tau:step in $tau:dot/child::supportIncident
              tau:copy-restricted-subtree($tau:p1, $tau:step))
     for $tau:p2 in tau:all-const-periods($tau:p, $tau:par1) return
       tau:associate-timestamp($tau:p2,
                               count(tau:snapshot($tau:par1, $tau:p2/@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 5.6. The Result of Copy-Based Per-Expression Slicing

of the tree is evaluated, the intermediate result in copy-based slicing is shown in Figure 5.7(b). Copy-based slicing makes a copy of the relevant portion with the correct timestamp. In-place slicing returns the original sub-tree with an actual timestamp as shown in Figure 5.7(c).



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

FIGURE 5.7. Intermediate results for per-expression slicing

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. ⟨Q⟩ ::= ⟨QueryProlog⟩ ⟨QueryBody⟩

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

   ```
   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.

2. ⟨QueryProlog⟩ ::= (⟨NamespaceDecl⟩
   | ⟨XMLSpaceDecl⟩
   | ⟨DefaultNamespaceDecl⟩
   | ⟨DefaultCollationDecl⟩
   | ⟨SchemaImport⟩)* ⟨FunctionDefn⟩*

   Among the non-terminals on the right-hand-side, only ⟨FunctionDefn⟩ need to be translated.

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

   inp ⟦⟨FunctionDefn⟩⟧ p =
   `define function` ⟨FuncName⟩ `(` (*inp* ⟦⟨ParamList⟩⟧ p)? `) as 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. ⟨QueryBody⟩ ::= ⟨ExprSequence⟩?
   ⟨ExprSequence⟩ ::= ⟨Expr⟩ (`,`⟨Expr⟩)*
   ⟨Expr⟩ ::= ⟨ForExpr⟩ | ⟨LetExpr⟩
   ⟨ForExpr⟩ ::= (⟨ForClause⟩ ⟨OrderByClause⟩? `return`)* ⟨TypeswitchExpr⟩
   ⟨ForClause⟩ ::= `for` ⟨SequenceType⟩? `$`⟨VarName⟩ `in` ⟨Expr$_1$⟩

⟨OrderByClause⟩ ::= `order by` ⟨Expr$_2$⟩ ⟨OrderModifier⟩

inp ⟦⟨ForExpr⟩⟧ p =

```
let $tau:s := inp ⟦⟨Expr1⟩⟧ p
for $tau:i in (1 to (count($tau:s) div 2))
let $tau:vi := 2 * $tau:i - 1
let $tau:v := item-at($tau:s, $tau:vi)
let $tau:p := item-at($tau:s, $tau:vi+1)
let $⟨VarName⟩ := ($tau:v,$tau:p)
let $tau:s1 := inp ⟦⟨Expr2⟩⟧ $tau:p
for $tau:i1 in (1 to (count($tau:s1) div 2))
let $tau:vi1 := 2 * $tau:i1 - 1
let $tau:v1 := item-at($tau:s1, $tau:vi1)
let $tau:p1 := item-at($tau:s1, $tau:vi1+1)
order by $tau:v1
return inp ⟦⟨TypeswitchExpr⟩⟧ $tau:p1
```

The variable ⟨VarName⟩ is bound to an (item, timestamp) pair instead of a single item in *cb* ⟦ ⟧. The `order by` 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.

4. ⟨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.

5. ⟨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. Similarly, the function `get-actual-items()` returns the items in the odd position

as a sequence. The function `interleave()` takes two sequences as inputs and interleaves them as one sequence.

6. ⟨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
   ```

7. ⟨ValueExpr⟩ ::= ⟨ValidateExpr⟩ | ⟨CastExpr⟩ | ⟨Constructor⟩ | ⟨StepExpr⟩
   ⟨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))
   ```

8. ⟨CastExpr⟩ ::= `cast as` ⟨AtomicType⟩ (⟨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 $tau:v := cast as ⟨AtomicType⟩ tau:get-actual-items($tau:ss)
   return tau:interleave($tau:v, get-periods($tau:ss))
   ```

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

   Among the non-terminals on the right-hand-side, the mapping of ⟨XmlComment⟩ and ⟨XmlProcessingInstruction⟩ 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 computed element constructor and computed attribute constructor 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, $tau:s) return
  (element {tau:copy-restricted-items(
                  tau:sequence-in-period($tau:s, $tau:p))}
   {tau:copy-restricted-items(inp ⟦⟨ExprSequence⟩⟧ $tau: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, $tau:s) return
  (attribute ⟨QName⟩
   {tau:copy-restricted-items(tau:sequence-in-period($tau:s,
                                          $tau:p))}, $tau:p)
```

10. ⟨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 $tau:s/timeVaryingAttribute[@name=⟨NameTest⟩]
   where tau:overlaps($tau:ta, $tau:p1) return
     ($tau:ta, tau:intersection($tau:ta, $tau:p1)))
```

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

inp ⟦⟨Literal⟩⟧ p = (⟨Literal⟩, p)

inp ⟦$⟨VarName⟩⟧ p =

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

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

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

The mapping of 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 $tau:p in tau:all-const-periods2(p, ($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 5.5 is mapped to the XQuery query in Appendix J. We put it in the Appendix because it is too long to fit in a figure. 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.

5.3 Idiomatic Slicing

Idiomatic slicing applies to both copy-based and in-place 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 one line of let-for expression. 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 of copy-based slicing, the data is time-sliced and the valid timestamps are propagated to the lower level nodes by copying the valid subtree. Since `let` and `for` expressions both time-slice the expression appearing in them, there are a lot of time-slices generated. The intermediate results are copied at each step. In the example query, each variable `$tau:dot` is copied at least twice. Excessive copying can potentially lead to bad performance.

To avoid the extra slicing and copying, 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 query in Figure 5.8 using the copy-based idiomatic slicing.

The auxiliary function `seq-path()` is defined in Appendix C. It returns the sequenced query results of a path expression. This function traverses the subtree rooted at the context node. When it visits a node, it checks whether the node is a target node. If the node is a target node, it copies and returns the valid subtree of the node. Thus, copying occures only when the node is found instead of at each step. Compared with the query in Figure 5.6, the length of the query body is reduced

```
{-- the normalized result:
validtime avg(for $c in
               (for $tau:dot in document("CRM.xml")//customer return
                 if ($tau:dot/attribute::supportLevel = "gold")
                 then $tau:dot
                 else ()) return
               count($c/supportIncident)) --}
let $tau:par :=
  (for $tau:i in
    (for $tau:i1 in tau:seq-path(tau:period("1000-01-01", "9999-12-31"),
                      document("CRM.xml")//customer, document("CRM.xml"))
     for $tau:p1 in tau:periods-of($tau:i1)
     let $tau:dot := tau:copy-restricted-subtree($tau:p1, $tau:i1) return
      for $tau:c in
       (for $tau:ta in $tau:dot/timeVaryingAttribute[@name="supportLevel"]
        return tau:copy-restricted-subtree($tau:p1, $tau:step))
      let $tau:p2 := tau:periods-of($tau:c)
      let $tau:b := element timeVaryingValue
                    {$tau:p2,
                     $tau:c/@value = "gold"}
      let $tau:s := tau:snapshot($tau:b, $tau:p3/@vtBegin) return
       if ($tau:s)
       then tau:copy-restricted-subtree($tau:p3, $tau:dot)
       else ())
   for $tau:p in tau:periods-of($tau:i)
   let $c := tau:copy-restricted-subtree($tau:p, $tau:i) return

    let $tau:par1 := tau:seq-path($tau:p, $c/supportIncident, $c)
    for $tau:p2 in tau:all-const-periods($tau:p, $tau:par1) return
     tau:associate-timestamp($tau:p2,
                             count(tau:snapshot($tau:par1, $tau:p2/@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 5.8. The Result of Copy-Based Idiomatic Slicing

dramatically by the copy-based idiomatic slicing. More importantly by reducing the number of nodes being copied, the performance of the mapped query can be improved potentially.

One limitation of `seq-path()` is it does not work when the path expression has predicates that contains time-varying nodes. In this case, the predicates are normalized first. In Figure 5.8, the first path expression is normalized before mapping, while the second one is mapped directly.

Idiomatic slicing can also be used to eliminate some of the unneeded slicing thus makes the mapped query more efficient. There are several situations in which idiomatic slicing applies. One is when a `let` expression binds a variable `$a` to a sequence, followed by a `for` expression that binds a variable `$b` to each of the items in `$a`. When the `for` expression is translated, there is no need to evaluate `$a` in sequenced semantics, because the evaluation period for `$a` does not change and the function `copy-restricted-subtree()` will do useless work on `$a`.

Idiomatic slicing also applies to in-place slicing. The advantage of in-place idiomatic slicing is that the length of the result query is shorter than that in in-place slicing. In-place idiomatic slicing use the auxiliary function `seq-path-inp()`, which is analogous to `seq-path-inp()`. This function pairs the target node with the timestamp representing its valid period instead of copying the target node.

5.4 Using the Schema

In all the time-slicing techniques we proposed, we didn't take schema into consideration since a schema is not always available and a schema is not required by XQuery. However, the schema information can help improve the mapping if it is available.

When evaluating a sequenced path expression, the τXQuery processor propagates the valid-time periods from the root to the target nodes since the target nodes may not have the valid-time periods, in which case, the target nodes inherit the valid-

time period of their nearest ancestor. However, if the τXQuery processor has the knowledge that the target nodes do have their own valid-time periods, it can save the propagation of the valid-time periods. This information can be easily found in the schema of the temporal XML document.

5.5 Comparison

We have proposed six 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 the resulting XQuery and/or the time complexity of that query by reducing copying.

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. It seems it is hard for maximally-fragmented slicing to be the best. One reason is that it has to traverse the document to find all the time points; the other is that it slices the whole document at all the time points. Consider a document with only the root node timestamped and there is only one version of the root. Suppose the temporal schema of the document is available. In

this case, the stratum can get the constant periods by looking at the root element only. Since there is only one constant periods, maximally-fragmented slicing will slice the document only once. The overheads mentioned above are not significant. Consider a query asks for all the sub-elements (specified as a wildcard) under a particular element over the entire timeline. 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 the root element and every leaf node timestamped. The root element has a very long valid period, while each leaf node has a very short valid period. There is one non-leaf element named `e`. A query asks for the element `e` favors this approach, because it can get the constant periods by looking at the root element only and time-slices the document only once. Maximally-fragmented slicing has to collect the time points of all the leaf nodes and 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 element 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 multiple 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.

Copy-based idiomatic slicing. Consider 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.

In-place idiomatic slicing. There are no cases that in-place idiomatic slicing outperforms others, but it can be one of the best. Consider the same document. Suppose the parent is the root element. When the query asks for all the child elements of the root element over the whole time line, in-place idiomatic slicing is as good as in-place slicing because it does not access more nodes than in-place slicing. Copy-based idiomatic slicing could be as good as in-place idiomatic slicing because it does not copy more nodes than in-place idiomatic slicing. Copy-based slicing is not good since it copies too many nodes. The other two slice the document many times.

The above comparison is based on our understanding of the six time-slicing techniques. Again, the performance tradeoffs depend on the way in which the underlying XQuery engine implements conventional XQuery queries. In next chapter, we empirically study the performance of the different time-slicing techniques.

5.6 Implementation

With the denotational semantics defined for each production, the implementation of the translator is straightforward. We first need a parser to scan the queries and build the parse tree for them. Each expression is a subtree in the parse tree. The queries are translated top down. The nested expressions are translated by calling the translator recursively. As an example, we show how ⟨LetExpr⟩ is implemented in copy-based slicing using pseudo code. The production and the denotational semantics are copied here for the convenience of the readers.

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

The subtree of the ⟨LetExpr⟩ in the parse tree is as shown in Figure 5.9.

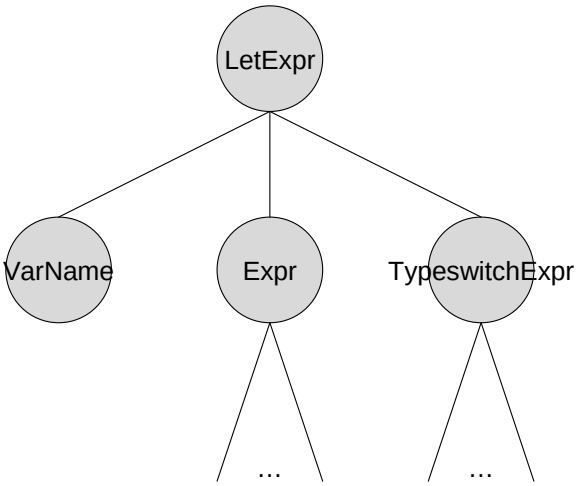


FIGURE 5.9. Parse tree of the ⟨LetExpr⟩

The denotational semantics below can be implemented by a function CBS_Let, which takes the parse tree node `LetExpr` and a valid time period as input parameters and returns a string, which is the XQuery code that evaluates the expression in sequenced semantics. The pseudo code of the function is shown in Figure 5.10.

$cb\,[\![\langle \text{LetExpr} \rangle]\!]\ p =$

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

The function CBS_Let needs to call function CBS_Expr and CBS_Switch to translate the two children nodes in the parse tree. Each of them returns a string that evaluates the corresponding expression in sequenced semantics based on the input valid time period. The strings then concatenated according to the definition of the denotational semantics. We didn't write the code for the translator. In the performance study, we manually translated the τXQuery queries to XQuery queries by following the denotational semantics.

```
FUNCTION: CBS_Let
          Returns a string which is the XQuery code for the ⟨LetExpr⟩.
INPUT:    LetExpr – a node in the parse tree
          p – the valid time period
OUTPUT:   A string which is the XQuery code

BEGIN
  result_code ="let $tau:s :=" + CBS_Expr(LetExpr.Expr, p);
  result_code = result_code +
              "for $tau:p in tau:const-periods(" + p.string + ", $tau:s)";
  result_code = result_code +
              "let $" + LetExpr.VarName +
              ":= tau:copy=restricted-subtree($tau:p, $tau:s)";
  result_code = result_code +
              "return" + CBS_Switch(LetExpr.TypeswitchExpr, "$tau:p");
  return result_code;
END
```

FIGURE 5.10. Pseudo code to translate ⟨LetExpr⟩

We didn't write the code for the translator. In the performance study, we manually translate the τXQuery queries to XQuery queries by following the denotational semantics.

Chapter 6
Performance Study for τXQuery

To compare the different time-slicing techniques presented in Chapters 3 and 5, we conducted a series of performance experiments. The objective is to find the most efficient time-slicing technique. The best way to do it is to run a benchmark on the stratum. Unfortunately, there are no temporal XML query benchmarks. Instead, we extended a non-temporal XML query benchmark to effect a temporal XML query benchmark. Section 6.1 introduces the non-temporal benchmark and describes how it was extended. The experimental setup and the results are discussed from Section 6.2 to Section 6.5. Finally, Section 6.6 summarizes the findings of the performance study.

6.1 Extending the Benchmark

There are several XML query benchmarks. These fall in two categories: micro benchmarks and application benchmarks [57]. *Micro benchmarks* are designed to test individual system components to isolate problems, measure, and thus, improve a particular component of an XML system. The Michigan Benchmark [40] belongs in this category. *Application benchmarks* measure the overall query performance of a DBMS. Benchmarks in this category include XMach-1 [7], XMark [41], XOO7 [9], and XBench [57]. The workloads of the first three benchmarks cover different functionalities, but leave out a number of XQuery features. XBench covers all XQuery functionality as captured by XML Query Use Cases [11]. Therefore, we chose XBench as the base non-temporal benchmark.

Database applications are charaterized along two dimensions: application characteristics and data characteristics. Application characteristics indicate whether

they are *data-centric* (*data-oriented*) or *text-centric* (*document-centric* or *document-oriented*) [22]. In data-centric (DC) XML, the set of XML vocabularies represent data that is more tightly structured than in text-centric (TC) XML. Text-centric XML is used when authoring loosely structured natural language documents. In terms of data characteristics, two classes are identified: *single document* (SD) and *multiple document* (MD). In single document case, the database consists of a single document with complex structures, while the multiple document case covers those databases that contain a set of XML documents. Thus, XBench consists of four different applications that cover DC/SD, DC/MD, TC/SD, and TC/MD respectively. We focused on DC/SD application since valid-time concepts are more applicable to it than to the other three applications. The DC/MD application has a large number of XML documents, each of which contains the information of an online order. Since placing an order is an event occuring at an instant, it is not realistic to add periods of validity to such informantion. The TC/SD XML document is a dictionary with numerous word entries and the XML documents in TC/MD application are a set of XML articles. The valid-time concept does not fit in these two applications. Thus, we chose the DC/SD application of XBench as our starting point.

The DC/SD XML document is called `catalog.xml`; this document stores an online catalog of a book store. The root element is `catalog` which has a sequence of `item` elements embedded in it. Each `item` stores the information for a book. The visual representation of the tree structure of all element types is shown in Figure 6.1 (excerpted from the XBench technical report [57]). The rectangles refer to element types; solid rectangles mean element types are mandatory while dotted ones mean they may or may not exist in a given document. By default, only one instance of a particular element type can appear in a real document, unless otherwise specified under rectangle. "(1..n)" means element types are mandatory and can appear up to n times. "(0..n)" means element types are not mandatory and can appear up to n times. If an element has attributes, they appear on the top right corner of the

rectangle where the element is located.

The schema and the data generated by XBench are non-temporal, as are the queries in the workload. We need to extend both the data and the queries of XBench. First, the time-varying elements were identified: `item`, `mailing_address` and `phone_number` (descendants of `author`), `related_item`, and `quantity_in_stock`. There were two steps to generate the temporal XML document. The first step was to generate a non-temporal XML document as the initial snapshot of the temporal XML document. In the second step, we implemented a program that simulated the evolving of the catalog. We used the generator of XBench to generate a non-temporal XML document with the default size 100MB. All the `item` elements in this document were divided into two groups. One group formed the initial snapshot of the temporal XML document (about 20MB); the other group served as the repository in the simulation step. Assume the starting time of the temporal document was st. Every time-varying element in the initial snapshot was associated with the valid time period $[st, forever)$. The simulation program set the current time as st, then it generated one interval i based on the predefined update frequency (the distribution of the valid period length of each snapshot will be specified in Section 6.2) of the document. The program advanced the clock by i. The current time was set as $st + i$. At each time point, the program randomly chose some of the time-varying elements from the current snapshot and randomly chose an operation (insert, delete, or update) for each of the chosen elements. For example, at some time point, the program decided to delete two `item` elements, update ten `quantity_in_stock` elements, and insert one `related_item` element. If a chosen element was not eligible for some operation, the program would randomly choose an operation from the applicable operations. For example, `quantity_in_stock` cannot be inserted to or deleted from an existing `item`. The only applicable operation is update. A delete operation set the ending time of the element to the current time; an insert operation inserted an element with the valid period $[current\_time, forever)$; and an update operation was a delete of the current

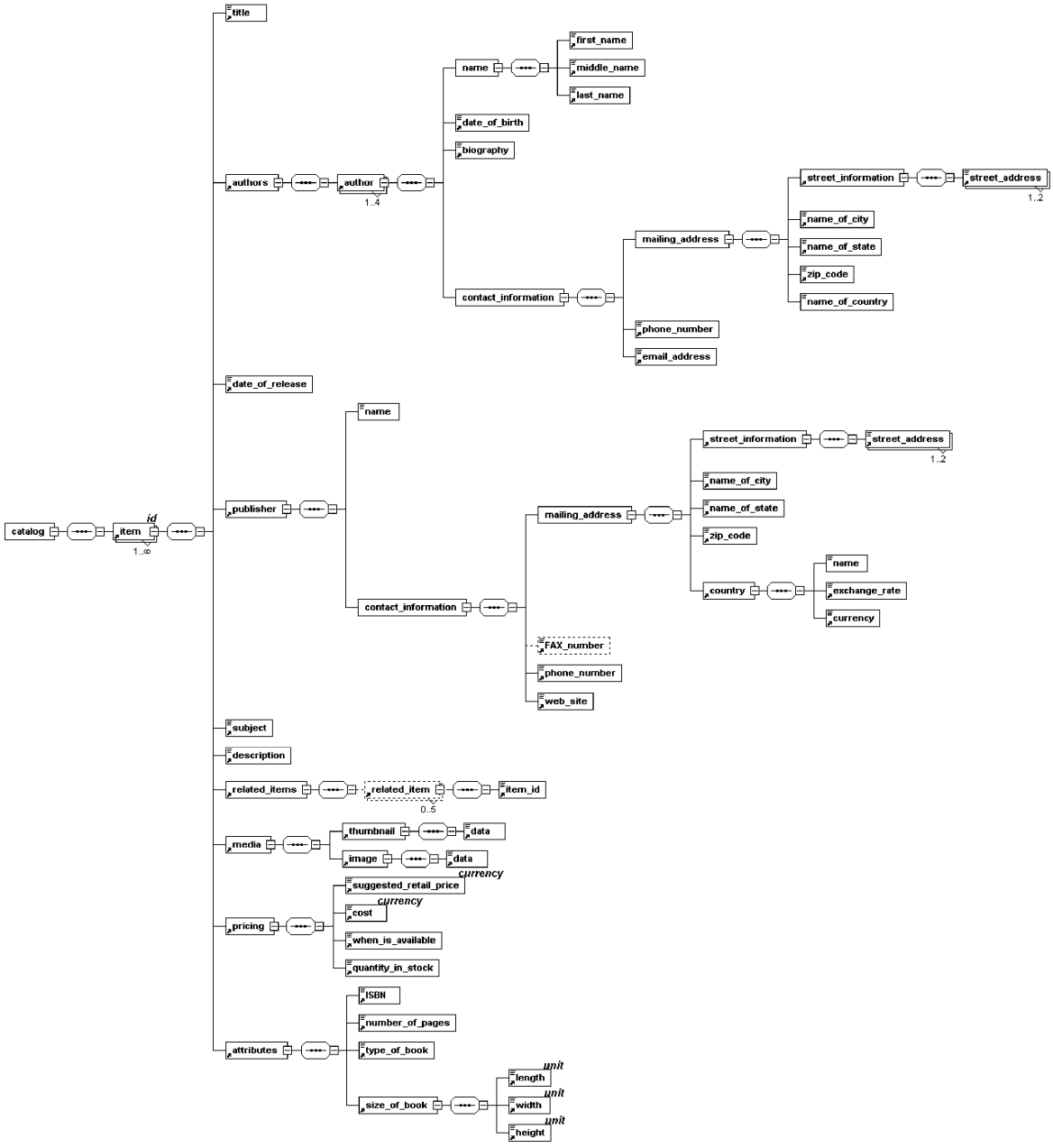


FIGURE 6.1. Schema diagram of DC/SD (Catalog)

element with an insertion of a new element. Both insersion and update needed new data which came from the repository obtained from the first step. Once some data has been added to the temporal XML document, it was removed from the repository. The program advanced the clock until the required number of changes have been generated. In this way, we obtained a temporal XML document.

We extended all the queries in the workload by adding the keyword `validtime` and an optional period $[st, et]$ in front of each query. How the period is decided is explained in Section 6.2. Thus, all the queries were extended to sequenced queries.

We ran the sequenced queries against the temporal XML document. The experimental setup is discussed in Section 6.2. We have used temporal XBench to test the performance of different time-slicing techniques. Instead of implementing the whole stratum, we manually mapped the sequenced queries to semantically equivalent XQuery queries by using different time-slicing techniques, which were then evaluated on the XQuery engine Galax. The applicability of the different techniques to the queries in the benchmark is discussed in Section 6.3. Sections 6.4 and 6.5 demonstrate the results of running the queries over periods of different lengths.

6.2 Experimental Setup

All experiments were conducted on a 2.4GHz Pentium 4 machine with 2GB main memory and two 40GB EIDI disk drives running Red Hat Linux 9.0 with kernel version 2.4.20. We chose Galax [31] as the underlying XQuery engine. The reason is that it is the only one we would find that supports user-defined functions. Since user-defined functions are used extensively to implement auxiliary functions, it is more easy using Galax than using another query engine with the user-defined functions inlined. Galax is a main-memory XQuery processor. Therefore, the majority of the elapsed time we measured in our study is CPU time.

The parameters used when the temporal data was generated are as follows. The

starting time of the document is `2002-01-01`. The length of the valid period of each snapshot is uniformly distributed between 1 to 30 days. The document stores the catalog history from `2002-01-01` to now. The total number of changing points in this document is 63 and the last changing time is `2004-06-12`. At each changing point, the number of elements changed is uniformly distributed between 1 and 5. The size of the temporal XML document is 25.7MB. The total number of nodes in the document is 3,152,577.

6.3 Applicability

The queries in XBench DC/SD workload are provided in Appendix K. There are sixteen queries in the workload (Q1–Q12, Q14, Q17, Q19, Q20). Note that the numbers are not continuous. Queries with missing numbers, such as Q13 and Q15, exist in the workloads of some other applications, but not in that of DC/SD application. There are six time-slicing techniques to be used, among which maximally-fragmented slicing and in-place slicing apply to all the queries. Selected-node slicing is not applicable to Q8 due to its wildcard. Copy-based slicing techniques (idiomatic and non-idiomatic) do not work for Q4 and Q20 because Q4 compares the relative positions of two elements (`input()/catalog/:item[.<<$item]`) and Q20 queries the parent node (`$size/../../title`), which needs the internal identity of nodes unchanged. Similarly in-place idiomatic slicing is not applicable to Q20 since idiomatic slicing cannot find the parent information. Hence, a total of 90 XQuery queries were produced and ran on the temporal document.

6.4 Querying over the Whole Timeline

In the first experiment, we ran all the sequenced queries over the whole timeline. To do this, we added the keyword `validtime` before each query without the optional period. The query should return all the data in the history that satisfies the query

predicates. For a given document and query, this is the worst-case performance for all the time-slicing techniques because in general, more data needs to be accessed when the query period is long. The elapsed time of different queries using different techniques are grouped by queries in Figure 6.2. The right-most group is the average elapsed time of different techniques over all the queries.

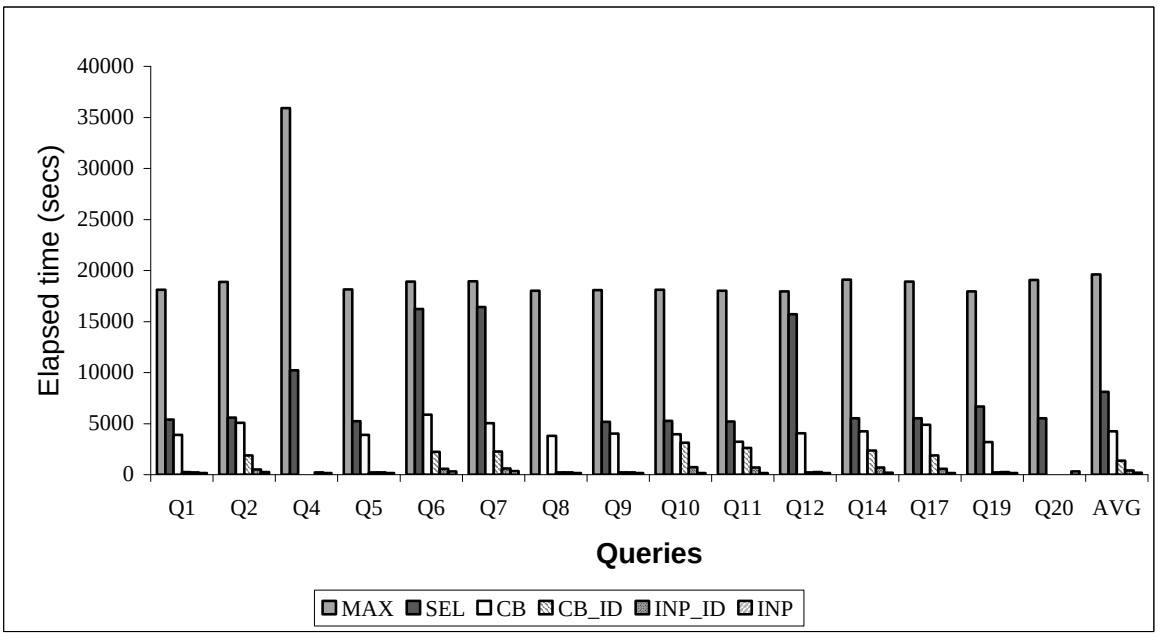


FIGURE 6.2. Query over the whole timeline

The results of all the queries are shown in Figure 6.2 except for those of Q3. The reason is that the results of all the techniques for Q3 are far poorer than the results for other queries. If we drew the results of Q3 in this figure, the results of other queries would not be seen clearly. Therefore, we show the numbers of Q3 in Table 6.1. Q3 groups all the items by the publishers and count the number of items in each group.

In Figure 6.2, MAX (maximally-fragmented slicing) performed worst for all the queries since it slices the whole document at all the changing points. SEL (selected-node slicing) had better performance than MAX because SEL slices the document

MAX	SEL	CB	CB_ID	INP_ID	INP
>5 days	375,000	250,000	71,400	71,900	12,600

TABLE 6.1. Results for Q3 (secs)

at only the changing points of selected nodes. More specifically, the cost of SEL is about one third of the cost of MAX for most queries, except for Q6, Q7, and Q12, for which the cost of SEL is close to the cost of MAX. This can be observed in the following numbers. As mentioned in Section 6.2, this document has 63 changing points. Therefore, MAX sliced the document 63 times and executed the non-temporal queries 63 times. SEL sliced the document 55 times for Q6, Q7, and Q12, and 18 times for the rest of the queries.

To better illustrate the performance of the remaining techniques, we remove the bars of MAX and SEL from Figure 6.2 and show the rest in Figure 6.3. While CB (copy-based slicing) performed worst, INP (in-place slicing) had the best performance. The average elapsed time of INP was 200 secs, which is only one twentieth of that of CB (4200 secs). The reason for this difference in performance is that CB copies the sub-trees of the document when evaluating each embedded expression while INP never copies. As an example, the result query of Q1 translated by CB is shown in Figure 6.4. The function `copy-restricted-subtree()` copies the valid subtree of the input tree. This function appears at each step of the path expression. Figure 6.5 shows the result query of Q1 translated by INP. There is no copying in INP-translated query.

CB_ID (copy-based idiomatic slicing) performed better than CB in all cases. CB_ID does not normalize the path expression. Thus, it only copies the result when the whole path expression is evaluated. CB copies the result when each step of the path expression is evaluated. To determine whether the copy operation is the major factor for the poor performance of CB, we calculated the number of nodes copied in each query by CB and CB_ID. The results are shown in Figure 6.6. When CB_ID

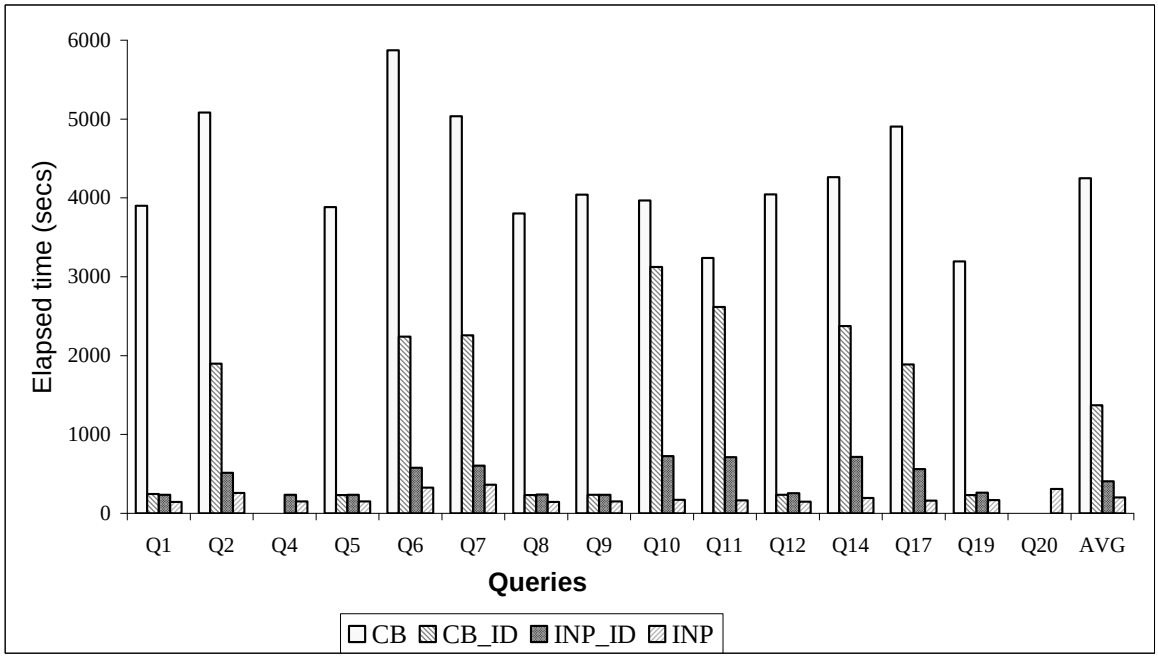


FIGURE 6.3. Query on the whole timeline

did not copy any nodes or when it copied a small number (< 500) of nodes (i.e., in Q1, Q5, Q8, Q9, Q12, and Q19), it had almost the same performance with INP_ID. When CP_ID copied a large number of nodes, the performance difference between non-idiomatic and idiomatic version correlated with the number of nodes copied.

INP was always better than its idiomatic version (INP_ID) in terms of performance. Recall the difference between them is in the way they handle path expressions. INP_ID evaluates sequenced path expressions without normalization. INP evaluates path expressions step by step. The queries obtained from INP_ID tend to be short. For example, the result query of Q1 translated by INP_ID, shown in Figure 6.7 is shorter than the query in Figure 6.5. This made us conjecture that INP_ID would have better performance than INP. On the other hand, INP_ID calls a function `tau:seq-path-inp()` to evaluate a sequenced path expression. This function traverses the sub-tree rooted at the context node to find the target node of the sequenced path expression. In the worst case, it needs to traverse the whole sub-tree.

```
let $tau:s := tau:copy-restricted-subtree(tau:period("1000-01-01",
                                  "9999-12-31"), document("CRM.xml"))
let $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 $tau:i1 in tau:copy-restricted-subtree($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
    for $tau:i in (for $tau:step in $tau:dot/catalog return
                     tau:copy-restricted-subtree($tau:p1, $tau:step))
    for $tau:p in tau:periods-of($tau:i)
    let $tau:dot := tau:copy-restricted-subtree($tau:p, $tau:i) return
      for $tau:i1 in (for $tau:step in $tau:dot/:item return
                       tau:copy-restricted-subtree($tau:p, $tau:step))
      for $tau:p1 in tau:periods-of($tau:i1)
      let $tau:dot := tau:copy-restricted-subtree($tau:p1, $tau:i1)
      return
        for $tau:c in (for $tau:ta in $tau:dot1/timeVaryingAttribute
                         where ($tau:ta/@name = "id") return $tau:ta)
        let $tau:p := tau:periods-of($tau:c)
        let $tau:b := element timeVaryingValue
                      {$tau:p,
                       $tau:c/@value = "I1"}
        let $tau:s := tau:snapshot($tau:b, $tau:p/@vtBegin) return
          if ($tau:s)
          then tau:copy-restricted-subtree($tau:p, $tau:dot)
          else ()
```

FIGURE 6.4. Query translated by CB from Q1

```
tau:apply-timestamp(
let $tau:s := (document("catalog.xml"), tau:period("1000-01-01", "9999-12-31"))
for $tau:p in tau:const-periods2(tau:period("1000-01-01", "9999-12-31"), $tau:s)
let $tau:sequence := tau:sequence-in-period($tau:s, $tau:p) return
  let $tau:s1 := tau:sequence-in-period($tau:sequence, $tau:p)
  for $tau:i1 in (1 to count($tau:s1) div 2)
  let $tau:vi1 := 2 * $tau:i1 -1
  let $tau:v1 := item-at($tau:s1, $tau:vi1)
  let $tau:p1 := item-at($tau:s1, $tau:vi1+1)
  let $tau:dot := ($tau:v1, $tau:p1) return
    let $tau:s2 := (let $tau:s := tau:get-actual-items($tau:dot)
                    let $tau:p := tau:get-periods($tau:dot)
                    where tau:overlaps($tau:p, $tau:p1) return
                      let $tau:p2 := tau:intersection($tau:p, $tau:p1)
                      for $tau:step in $tau:s/catalog
                      where tau:overlaps($tau:p2, $tau:step) return
                        ($tau:step, tau:intersection($tau:p2, $tau:step)))
    for $tau:i2 in (1 to count($tau:s2) div 2)
    let $tau:vi2 := 2 * $tau:i2 -1
    let $tau:v2 := item-at($tau:s2, $tau:vi2)
    let $tau:p2 := item-at($tau:s2, $tau:vi2+1)
    let $tau:dot := ($tau:v2, $tau:p2) return
      let $tau:s3 := (let $tau:s := tau:get-actual-items($tau:dot)
                      let $tau:p := tau:get-periods($tau:dot)
                      where tau:overlaps($tau:p, $tau:p2) return
                        let $tau:p3 := tau:intersection($tau:p, $tau:p2)
                        for $tau:step in $tau:s/:item
                        where tau:overlaps($tau:p3, $tau:step) return
                          ($tau:step, tau:intersection($tau:p3, $tau:step)))
      for $tau:i3 in (1 to count($tau:s3) div 2)
      let $tau:vi3 := 2 * $tau:i3 -1
      let $tau:v3 := item-at($tau:s3, $tau:vi3)
      let $tau:p3 := item-at($tau:s3, $tau:vi3+1)
      let $tau:dot := ($tau:v3, $tau:p3) return
        for $tau:c in (let $tau:s := tau:get-actual-items($tau:dot)
                       let $tau:p := tau:get-periods($tau:dot)
                       where tau:overlaps($tau:p, $tau:p3) return
                         let $tau:p4 := tau:intersection($tau:p, $tau:p3) return
                           (for $tau:a in $tau:s/@id return ($tau:a, $tau:p4),
                            for $tau:ta $tau:s/timeVaryingAttribute[@name="id"]
                            where tau:tau:overlaps($tau:p4, $tau:ta) return
                              ($tau:ta, tau:intersection($tau:p4, $tau:ta))))
        let $tau:s := (item-at($tau:c, 1)="I1" or item-at($tau:c, 1)/@value="I1",
                       item-at($tau:c, 2))
        for $tau:p in tau:const-periods2($tau:p2, $tau:s) return
          if (tau:get-actual-items(tau:sequence-in-period($tau:s, $tau:p)))
          then tau:sequence-in-period($tau:dot, $tau:p)
          else ())
```

FIGURE 6.5. Query translated by INP from Q1

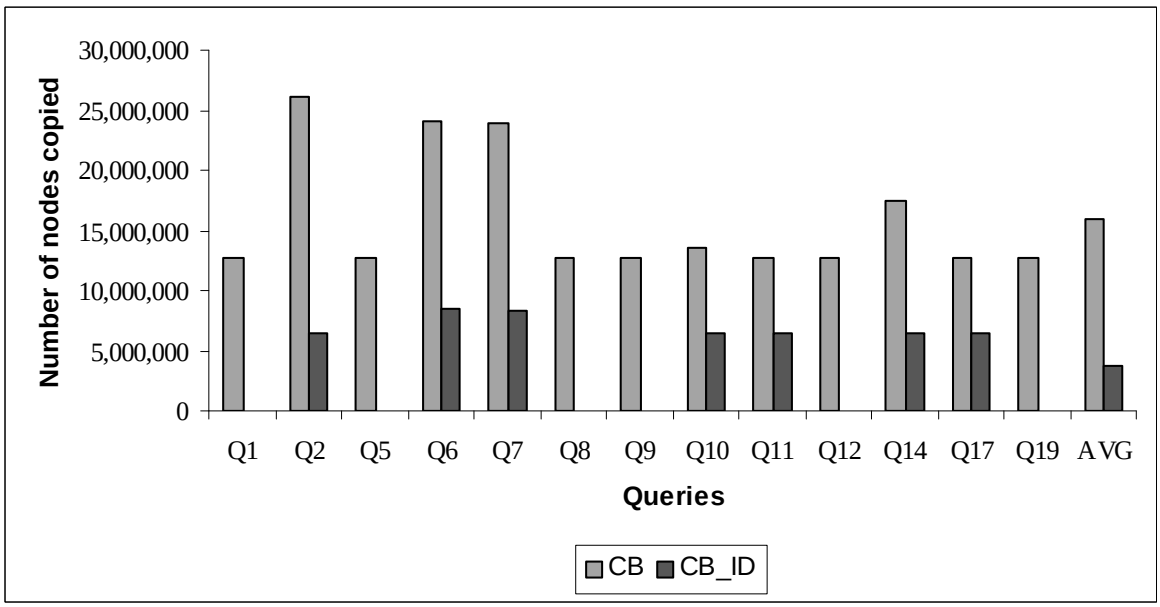


FIGURE 6.6. Number of nodes copied

If the size of the sub-tree is large, the cost of traversal is high. The good side about INP is that it only accesses the nodes on the path. Target nodes can be found without accessing all the nodes. This is the reason that INP performed better than its idiomatic version.

```
tau:apply-timestamp(
    tau:seq-path-inp(tau:period("1000-01-01", "9999-12-31"),
                     document("catalog.xml")/catalog/:item[@id="I1"],
                     document("catalog.xml")))
```

FIGURE 6.7. Query translated by INP_ID from Q1

6.5 Querying over a Short Period

In this experiment, we ran all the sequenced queries over a short period, that of one day. We placed `validtime [2003-01-01, 2003-01-01]` in front of each query, to see the effect of the length of query period. The results are shown in Figure 6.8.

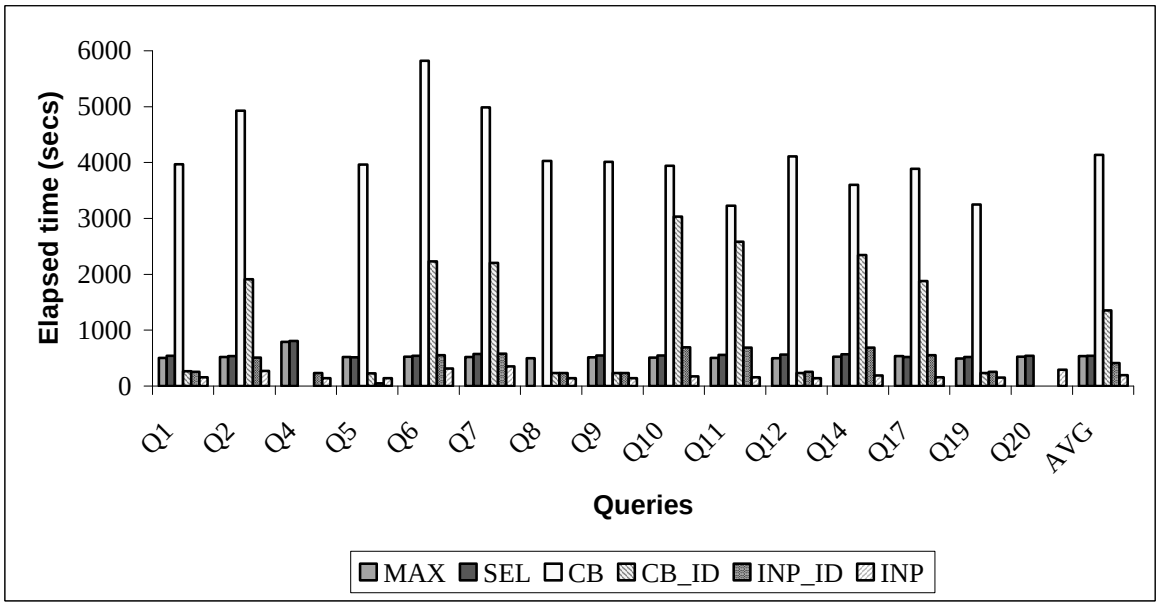


FIGURE 6.8. Querying on a short period

MAX and SEL had dramatic improvement of their performance as compared with the performance in Section 6.4. The reason is that both of them need to slice the document only once in this experiment, while in the previous experiment, they both sliced the document multiple times (18 to 63). In the short period experiment, SEL was a little bit worse than MAX because when it collects the time points, it needs to judge whether the encountered node is in the selected set.

The performance of other slicing techniques is similar to that of the whole timeline experiment. INP and INP_ID did not take advantage of short query period because they always keep the original sub-tree even if only a portion of the sub-tree is valid. Since CB prunes the invalid branches, it should perform better in short period queries than in long period queries. However, the difference is not seen in this case because the number of nodes pruned was far less than the number of nodes copied. As mentioned in Section 6.2, no more than five elements in the document change. Since most elements remain unchanged most of the time, data valid on a particular day has

almost the same size with the whole data set.

INP was still the best. In most cases, MAX and SEL had comparable performance with INP_ID.

6.6 Summary

The empirical observations show that the time-slicing techniques are applicable to most of the queries in a sequenced version of XBench. Among them, MAX and INP work for all queries.

The length of the query period has a significant impact on the performance of the two slicing techniques that slice the data at the document level. The shorter the query period, the fewer the slices, therefore, the better the performance of MAX and SEL. When the query period is long enough, the performance of them are unacceptable.

Excessive copying led to the bad performance of CB. Idiomatic slicing is not always better than non-idiomatic counterpart because it accesses extra nodes when traversing the sub-tree rooted at the context node. In all the cases, non-idiomatic in-place slicing was the best and therefore is recommended.

The data and queries used in the experiments do not cover all the cases. We expect the length of the query period impacts the performance of copy-based (non)idiomatic slicing because it prunes the invalid subtree. However, our data does not change frequently. Thus most part of the document is valid at most time. In this case, copy-based slicing needs to keep most of the subtree. We did not demonstrate the cases that favor maximally-fragmented slicing and selected node slicing because these are very extreme cases that do not often occur in reality.

For all the queries in the workload, the temporal semantics can be written in XQuery queries directly. We didn't compare the performance of a well-written temporal query using only XQuery and the performance of the best automatically translated query. The reason is that in our translator, the schema information is not considered

while a well-written XQuery took the schema into consideration. Thus we expect a well-written XQuery be better than the automatically translated queries. However, this needs to be verified in the future.

We didn't compare the performance of the mapped queries with the non-temporal queries. This case is less important than the comparison we discussed in the last paragraph. Temporal queries are more complex than the corresponding non-temporal queries. So there is no doubt that temporal queries are worse than the non-temporal ones in performance.

So far we have shown the temporal support of XQuery, its implementation strategy and techniques. In next chapter, we turn to the temporal support of SQL/PSM.

Chapter 7
Temporal SQL with PSM

From this chapter, we move our attention to another popular query language, SQL [49]. As the standard query language for relational database, SQL is widely used in database applications. Previous research on temporal SQL [47] focused on data definition language and data manipulation language. Persistent Stored Module (PSM), a part of SQL standard, was never addressed in any research work on temporal SQL. In this chapter, we present our solution of integrating PSM with one of the proposed temporal extension to SQL. Section 7.1 briefly introduces the basic concepts and terminology in SQL/PSM. The subsequent section summarizes previous research on temporal data definition language and temporal data manipulation language. Two questions about combining SQL temporal and PSM are raised in Section 7.3 and the solution is provided.

7.1 SQL/PSM

When hearing the word "SQL", most people think of `SELECT`, `INSERT`, `DELETE`, and `UPDATE` statements. In fact, SQL standard has more language provisions than just these four statements. Persistent stored modules (PSM) and control statements are both important and useful components of the standard SQL [37, 49].

PSM are stored programs that are compiled and stored in some schema, and run at the SQL server side. PSM consists of stored procedures and stored functions, which are collectively called *stored routines*. As in any other programming language, the difference between procedures and functions is that functions have exactly one explicit return value while procedures could have output parameters but not an explicit

return value. Stored routines can be written in either SQL or one of the programming languages with which SQL has defined a binding (Ada, C, COBOL, Fortran, etc.). Stored routines written entirely in SQL are called *SQL routines*; stored routines written in other programming languages are called *external routines*. In programming languages such as C, the control statements are used throughout most programs. The SQL standard [49] also provides control statements, which usually appear in stored routines.

Each commercial DBMS has its own idiosyncratic syntax and semantics of PSM. For example, the language PL/SQL created by Oracle supports PSM and control statements. Microsoft's Transact-SQL provides extensions to standard SQL that permit control statements and stored procedures. IBM, Sybase, and Informix all have their own implementation of the features similar to those in SQL/PSM.

The advantages of writing stored routines in database applications have been described in Chapter 1. Here we look at an example in which we create a stored routine written in SQL and invoke it in a query. Suppose in a music/video store application, there are two relations `movie_titles` and `music_titles` that keep the information of movie videos and music records sold in the store, respectively. Figure 7.1 shows the schemas of the two relations. In Figure 7.2, function `discount_price()` takes a movie

```
movie_titles(movie_id, movie_title, movie_price)
music_titles(music_id, music_title, music_price)
```

FIGURE 7.1. Schemas of the two relations

title as input and returns the discount price of this movie. Query Q1 in Figure 7.3 returns the music records that have higher price than the discount price of the movie "Star Wars". Q1 calls the function `discount_price()` in its where clause. Of course, this query can be written without utilizing stored functions. But our objective here is to show how a stored routine can be used to accomplish the task.

```
CREATE FUNCTION discount_price(title CHARACTER(10))
    RETURNS DECIMAL(5,2)
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE regular_price DECIMAL(5,2);
    SET regular_price = (SELECT movie_price
                         FROM movie_titles
                         WHERE movie_title = title);
    IF (regular_price > 10) THEN
      RETURN regular_price * 0.8;
    ELSE
      RETURN regular_price;
  END;
```

FIGURE 7.2. Function `discount_price()`

```
Q1: SELECT music_title
    FROM music_titles
    WHERE music_price > discount_price("Star Wars");
```

FIGURE 7.3. A query calling `discount_price()`

7.2 SQL/Temporal

SQL/Temporal [47] was proposed as a part of the SQL:1999 standard [49]. To ensure upward compatibility and temporal upward compatibility, SQL/Temporal classifies temporal queries into three categories: *current query*, *sequenced query*, and *nonsequenced query* [47]. This is analogous to the three kinds of queries in τXQuery. The name "nonsequenced" implies all the queries that are not sequenced queries. In fact, it does not include current queries, which are not sequenced queries, either. That is why we use a more accurate name "representational" in τXQuery for this kind of query. However, when we talk about SQL/Temporal in this dissertation, we would like to follow the convention of the original SQL/Temporal proposal. Please keep in mind "nonsequenced" queries in SQL/Temporal are actually representational queries.

Current queries only apply to the current state of the database. Sequenced queries apply independently to each state of the database over a specified temporal period. Users don't need to explicitly manipulate the timestamps of the data when writing either current queries or sequenced queries. Nonsequenced queries are those complex temporal queries that are not in the first two categories. Users explicitly manipulate the timestamps of the data when writing nonsequenced queries. Two additional keywords are used to differentiate the three kinds of queries from each other. Queries without temporal keywords are considered as current queries. Sequenced queries and nonsequenced queries have the temporal keywords `VALIDTIME` and `NONSEQUENCED VALIDTIME`, respectively, in front of the conventional queries.

Similar to τXQuery, the implementation of SQL/Temporal can be done using a stratum approach. Implementing nonsequenced queries in the stratum is trivial. Current queries are special cases of sequenced queries. SQL/Temporal defined temporal algebra operators for sequenced queries. When the stratum receives a temporal query, it first maps it into temporal algebra, then into conventional SQL.

Let's look at an example temporal query. Suppose the two relations `movie_titles` and `music_titles` mentioned in Section 7.1 are now temporal tables with valid-time support. That is, each row of the relations is associated with a valid-time period. Now a query asking for the history of the price of the movie "Star Wars" would be written as the following sequenced query.

```
Q2: VALIDTIME SELECT movie_price FROM movie_titles
    WHERE movie_title = "Star Wars";
```

When this query is evaluated on a temporal DBMS that is built upon a conventional DBMS, it is mapped to a conventional SQL query, which is then evaluated on the conventional DBMS. The result of this query is a temporal relation, each row of which is a price associated with a valid period.

7.3 Putting SQL/Temporal and PSM Together

While SQL/Temporal extended the data definition statements and data manipulation statements in SQL, it never addressed temporal PSM. The design of SQL/Temporal facilitates the migration of legacy applications to temporal DBMS. Assume we have a nontemporal application that contains the function `discount_price()` in Figure 7.2. Now we intend to migrate the application to a temporal DBMS, in which case, the underlying relations are temporal relations. There are two questions we want to ask before the migration. The first question is "Can the function `discount_price()` be called by a temporal query and if so, what is the semantics?" The second question is related to the implementation of temporal PSM: "How can the semantics be implemented?" SQL/Temporal doesn't answer these two questions.

In this section, we first define the syntax and semantics of temporal PSM informally. Then the implementation of temporal PSM is discussed briefly. The details of the mapping techniques will be given in Chapter 8.

7.3.1 Syntax and Semantics of Temporal SQL/PSM

Temporal SQL/PSM is based on SQL/Temporal, which extends the syntax and semantics of SQL data definition statements and data manipulation statements. Temporal SQL/PSM merges SQL/Temporal with traditional SQL/PSM. There are no syntax extensions to Temporal SQL/PSM. No additional keywords are used when creating or invoking a stored routine. Therefore, the function `discount_price()` can be called in a temporal query as shown in Figure 7.4.

```
Q3: VALIDTIME SELECT music_title FROM music_titles
    WHERE music_price > discount_price("Star Wars");
```

FIGURE 7.4. A sequenced query calling `discount_price()`

The semantics of a stored routine depends on the invocation environment of the

routine. If a routine is called in a current (or sequenced or nonsequenced) query, the routine carries current (or sequenced or nonsequenced) semantics and every statement in this routine presents current (or sequenced or nonsequenced) semantics. This feature of stored routines eases the reuse of existing modules written in conventional SQL. Thus, Q2 returns the music records whose price was ever higher than the discount price of the movie "Star Wars" in the history. The result of Q2 is a temporal relation, each row of which consists of a music_title and a time period, during which the music record had a higher price than the movie. (Note that effectively the PSM function must return two things: a value and a validity time.)

Figure 7.5 and Figure 7.6 show the above function invoked in a current query and a representational query respectively. Q4 has the same semantics with the original SQL query, i.e., the music records whose price is currently higher than the current discount price of the movie "Star Wars". Q5 returns the music records whose price, now or in the past, is higher than the discount price of the movie "Star Wars", now or in the past.

```
Q4: SELECT music_title FROM music_titles
    WHERE music_price > discount_price("Star Wars");
```

FIGURE 7.5. A current query calling `discount_price()`

```
Q5: NONSEQUENCED VALIDTIME SELECT music_title FROM music_titles
    WHERE music_price > discount_price("Star Wars");
```

FIGURE 7.6. A nonsequenced query calling `discount_price()`

An alternative approach would allow each routine or each statement in the routine explicitly stated by the programmer to be current (or sequenced, or nonsequenced). This approach seems to provide more flexibility to users. However, we didn't choose this approach for two reasons. First, this approach needs new language constructs for temporal PSM to indicate the semantics for each routine and control statement.

Secondly the flexibility provided by this approach has nothing to do with the goal of this extension — easy migration of the legacy application.

7.3.2 Formal Semantics of Temporal SQL/PSM

We now define the formal syntax and semantics of temporal SQL/PSM query expressions. As in Section 3.4, we use a syntax-directed denotational semantics style formalism [50].

In SQL/Temporal, there are three kinds of SQL queries, in which PSMs can be invoked. The production of a temporal query expression can be written as follows.

⟨Temporal Q⟩ ::= (`VALIDTIME` (`[`⟨BT⟩`,` ⟨ET⟩`]`)$^{?}$| `NONSEQUENCED VALIDTIME`)$^{?}$⟨Q⟩

⟨Q⟩ is a regular SQL query. A query in SQL/Temporal is a current query by default, or a sequenced query if the keyword `VALIDTIME` is used, or a nonsequenced query if the keyword `NONSEQUENCED VALIDTIME` is used. Note that ⟨Q⟩ is a query that possibly invokes a stored function. The semantics of ⟨Temporal Q⟩ is expressed with the semantic function $TSQLPSM\,[\![]\!]$. $cur\,[\![]\!]$, $seq\,[\![]\!]$, and $rep\,[\![]\!]$ are the semantic functions for current query, sequenced query, and nonsequenced query. The traditional SQL semantics is represented by the semantic function $SQL\,[\![]\!]$.

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

$TSQLPSM\,[\![$`VALIDTIME [`⟨BT⟩`,` ⟨ET⟩`]` ⟨Q⟩$]\!] = seq\,[\![\langle \text{Q} \rangle]\!]$ `[`⟨BT⟩`,` ⟨ET⟩`]`

$TSQLPSM\,[\![$`NONSEQUENCED VALIDTIME` ⟨Q⟩$]\!] = rep\,[\![\langle \text{Q} \rangle]\!]$

The nonsequenced query has almost the same semantics as the regular SQL query. For most the language constructs, the following equation is true.

$rep\,[\![\langle \text{Q} \rangle]\!] = SQL\,[\![\langle \text{Q} \rangle]\!]$

The only exception is when `VALIDTIME` is used. This is a new construct in SQL/Temporal. It applies to a correlation name and returns the valid time period of the tuples. In this case, the semantic function is defined as follows.

rep ⟦`VALIDTIME(`⟨corr name⟩`)`⟧ =
`PERIOD(`⟨corr name⟩`.BEGIN_TIME,`⟨corr name⟩`.END_TIME)`

Current query and sequenced query need to be mapped. The mapping will be described in detail in Chapter 8.

7.3.3 Implementation of Temporal SQL/PSM

We chose the stratum approach for the same reason mentioned in Chapter 4. The stratum architecture of Temporal SQL/PSM is analogous to that of τXQuery except that the underlying query processor is a relational DBMS and the stratum translates Temporal SQL/PSM queries to conventional SQL/PSM queries. Although SQL/Temporal proposed the implementation of a stratum, the mapping techniques do not apply to Temporal PSM directly. The temporal relational algebra defined for temporal data statements cannot express the semantics of control statements and stored routines. Therefore, we need to use different techniques. Since XQuery has complex expressions and we have proposed the time-slicing techniques for τXQuery stratum, we expect that similar techniques (together with algebra based techniques) can be used to translate Temporal SQL/PSM queries to SQL/PSM queries.

The differences between the temporal relational data model and temporal XML data model renders some of the mapping techniques proposed in Chapter 5 meaningless. For example, in temporal relational model adopted by SQL/Temporal, each tuple is associated with exactly one timestamp, which applies to all the attributes values of this tuple. Thus, selected-node slicing has no counterparts for temporal SQL/PSM. The next chapter will examine in detail the other mapping techniques proposed in Chapter 5, as applied to temporal PSM.

Chapter 8
Mapping Techniques for Temporal PSM

In this chapter, we will define the detail of the formal semantics of current query and sequenced query. As this formal semantics is given by mapping the temporal query to semantically equivalent SQL/PSM, the mapping techniques will also be described at the same time. For current query mapping, we only describe a simple approach to define the semantics. A current query is a special case of a sequenced query. We will start from this special case, then focus on sequenced queries. Whatever techniques that can be used to map sequenced queries are also suitable for mapping current queries. Two techniques will be presented in this chapter, namely maximally-fragmented slicing and per-statement slicing. Two useful properties of the semantics will be discussed in this chapter.

8.1 Mapping Current Query

The semantics of a current query on a temporal database is exactly the same as the semantics of a regular SQL query on the current snapshot of the temporal database. The formal semantics of current query can be defined as follows.

$$cur\ [\![\langle \mathrm{Q} \rangle]\!]\ (r_1, r_2, ..., r_n) = SQL\ [\![\langle \mathrm{Q} \rangle]\!]\ \tau^{vt}_{now}(r_1, r_2, ..., r_n)$$

In the mapping above, r_1, r_2, ..., r_n denote tables that are accessed by the query $\langle \mathrm{Q} \rangle$. These are tables with valid time support. We borrow the temporal operator τ^{vt}_{now} from the proposal of SQL/Temporal [47]. τ^{vt}_{now} extracts the current snapshot value from one (or more) tables with valid time support.

Calculating the current snapshot of a table is equivalent to do a selection on the table. To map a current query (with PSM) in SQL, we just need to add one predicate

for each table to the where clauses of the query and queries inside the PSM. Assume r_1, ..., r_n are all the tables that are accessed by the current query. The following predicate needs to be added to all the where clauses.

r_1.begin_time <= CURRENT_TIME AND r_1.end_time > CURRENT_TIME AND

...

r_n.begin_time <= CURRENT_TIME AND r_n.end_time > CURRENT_TIME

As an example, the current query Q4 in Figure 7.5 should be mapped to the SQL query in Figure 8.1 and the current version of the function `discount_price()` should be mapped to the SQL query in Figure 8.2.

```
SELECT music_title FROM music_titles
WHERE music_price > cur_discount_price("Star Wars")
AND music_title.begin_time <= CURRENT_TIME
AND music_title.end_time > CURRENT_TIME;
```

FIGURE 8.1. The SQL query mapped from Q4

8.2 Maximally-Fragmented Slicing

The idea of maximally-fragmented slicing is similar to that of the one used to define the semantics of sequenced τXQuery in Section 3.4.3. We compute the constant periods first, then the SQL query is evaluated at each constant period. Section 8.2.1 gives the SQL statements for computing constant periods. SQL routines include functions and procedures. Functions can be invoked in both SQL data manipulation statements and control statements, while procedures can only be invoked inside a routine by a CALL statement. SQL data manipulation statements include SELECT statements and modification statements such as INSERT, DELETE, and UPDATE. We start with the mapping of queries that invoke functions. Nested invocation is also discussed in this section. Then we will look at SQL modification statements that invoke functions.

```
CREATE FUNCTION cur_discount_price(title CHARACTER(10))
    RETURNS DECIMAL(5,2)
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE regular_price DECIMAL(5,2);
    SET regular_price = (SELECT movie_price
                         FROM movie_titles
                         WHERE movie_title = title
                         AND movie_title.begin_time <= CURRENT_TIME
                         AND movie_title.end_time > CURRENT_TIME);
    IF (regular_price > 10) THEN
      RETURN regular_price * 0.8;
    ELSE
      RETURN regular_price;
  END;
```

FIGURE 8.2. Current version of the function in Figure 7.2

8.2.1 Constant Periods

To compute constant periods, all the timestamps in the input temporal relations 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 temporal data, and thus, of the result. Instead of storing the constant periods as a sequence of elements, we take the advantage of relational model and put the constant periods into a relation, each row of which has two attributes indicating the begin time and end time of a period.

Assume the relations involved in a query (including the relations appearing in the routines called by the query, which can be determined by a static analysis) are r_1, r_2, ..., r_n. The relation cp that stores the constant periods of the n relations is calculated by the following relational calculus expression.

$$cp(r_1, r_2, ..., r_n) = \{\langle bt, et\rangle | bt \in ts \wedge et \in ts \wedge bt < et \wedge \neg\exists t \in ts(bt < t < et)\}$$

This can be evaluated by a self-join of the single column relation ts, which stores all

the begin time and end time of the timestamps in the n input temporal relations. The following relational algebraic expression shows how to calculate ts. Basically, it is a union of the projection of the begin time column and end time column of all the temporal relations.

$$ts = \bigcup_{i=1}^{n} (\pi_{bt}(r_i) \cup \pi_{et}(r_i))$$

The relation `ts` can be obtained by running the following SQL query, assuming the valid time of each tuple is represented by `begin_time` and `end_time`.

```
SELECT begin_time AS time_point FROM  r_1
UNION
SELECT end_time AS time_point FROM  r_1
UNION
...
UNION
SELECT begin_time AS time_point FROM  r_n
UNION
SELECT end_time AS time_point FROM  r_n
```

Then the relation `cp` (which could also be expressed as a view) is then implemented by the SQL query below.

```
SELECT ts1.time_point AS begin_time, ts2.time_point AS end_time
FROM ts AS ts1, ts AS ts2
WHERE begin_time < end_time AND
      NOT EXISTS (SELECT time_point
                  FROM ts
                  WHERE time_point > begin_time
                  AND time_point < end_time)
ORDER BY begin_time
```

Once cp is computed, the query is evaluated against the snapshot of the input relations at the begin time of each tuple in *cp*.

8.2.2 SQL Queries Invoking Functions

An SQL query is written as one or more SQL SELECT statements. The syntax of an SQL SELECT statement is as follows.

⟨select statement⟩ ::=

SELECT ⟨select list⟩
FROM ⟨table reference list⟩
[WHERE ⟨search condition⟩]
[⟨group by clause⟩]
[HAVING ⟨search condition⟩]

Assume an SQL function F() is called in this select statement. The function can appear in several places in the select statement. It could appear as one of the columns in the select list or as a part of the search condition specified in either the where clause or the having clause. It could also appear in the table reference list if its return value is a collection type. A *collection* type in SQL standard is an array, which can be used as a derived table. In any case, the sequenced semantics of the select statement is as follows.

seq ⟦⟨select statement⟩⟧ =

SELECT *tsqlpsm* ⟦⟨select list⟩⟧, cp.begin_time, cp.end_time
FROM *tsqlpsm* ⟦⟨table reference list⟩⟧, cp
[WHERE *tsqlpsm* ⟦⟨search condition⟩⟧ AND
overlap ⟦*tables* ⟦⟨select statement⟩⟧, cp.begin_time⟧]
[⟨group by clause⟩, cp.begin_time]
[HAVING *tsqlpsm* ⟦⟨search condition⟩⟧]

A sequenced query always returns a temporal relation, i.e., each row of the relation is timestamped. Therefore, `cp.begin_time` and `cp.end_time` are added to the select list and `cp` is added to the from clause. A search condition is added to the where clause to ensure that tuples from every table overlaps the constant period. The semantic function *tables* ⟦ ⟧ returns an array of strings, each is a table reference appearing in the input query. The semantic function *overlap* ⟦ ⟧ returns a series of search conditions represented as a string. If there are n tables referenced in the statement, *overlap* ⟦ ⟧ returns n conditions, each of the form *tname*`.begin_time <= cp.begin_time AND cp.begin_time <` *tname*`.end_time`, where *tname* is the table name.

In the above mapping, four nonterminals need to be mapped (⟨select list⟩, ⟨table reference list⟩, and two ⟨search condition⟩s). If the select statement is not a nested query, the only aspect that need to be mapped in these four nonterminals are the function calls that occur in these nonterminals. A function call `F(`*par*`)` is mapped to `mf_F(`*par*`, cp.begin_time, cp.end_time)`, an extended version of `F`. In addition to the original parameters of `F()`, `mf_F()` passes the constant period into the function.

8.2.3 Functions Invoked in SQL Queries

The body of the definition of `F` also needs to be transformed. All the SQL queries inside `F` are mapped in `mf_F` to a temporal query at the input time point `cp.begin_time`. This translation is done by adding a condition of overlapping `cp` for each temporal table in the where clause. In the following mapping, semantic function *seqf* ⟦ ⟧ defines the transformation of a function definition.

seq ⟦⟨function definition⟩⟧ = *seqf* ⟦⟨function definition⟩⟧

seqf ⟦⟨parameter list⟩⟧ = ⟨parameter list⟩, `begin_time`, `end_time`

seqf ⟦⟨search condition⟩⟧ =

⟨search condition⟩ `AND` *overlap* ⟦*tables* ⟦⟨select statement⟩⟧, `begin_time`⟧

If there are SQL statements inside the function that modify data in some temporal tables (e.g., delete, insert, and update statement), these statements are mapped to sequenced statements that returns the results valid in `cp`. A sequenced modification to temporal tables can be done by a series of SQL modifications to the tables in conventional SQL; this mapping is given elsewhere [45]. An example of such kind of function will be shown later in this section.

The select statement could be a nested query, which has a subquery in either the from clause or the where clause. In this case, the subquery (a select statement) should be mapped to a temporal query at the time point `cp.begin_time`. Therefore, the mapping of a subquery is similar to the mapping of a query inside a function.

Assume a function f is mapped to f'. Function f can have routine invocations inside it. If another function f_1 is called inside f, the constant period passed into f' now is passed into f_1'. If a procedure is invoked inside f, it can be invoked only by a call statement in the following syntax.

⟨call statement⟩ ::= `CALL` ⟨procedure invocation⟩

The same constant period is passed to the procedure. The output parameters of the procedure remain unchanged.

In addition to a sequenced query, a function can be invoked in a sequenced modification statement. In this case, a cursor is defined for the constant period table `cp`. For each period in `cp`, a sequenced modification on this particular period is executed. The translation of the function is the same as that in a sequenced query.

As an example, let's look at the query Q2 mentioned in Section 7.3.1. In Figure 8.3, Q2 is translated to an SQL query that invokes the function `mf_discount_price()`, which is translated from `discount_price()`.

The reader may have noticed that the last parameter `end_time` is not used inside

```
SELECT music_title, cp.begin_time, cp.end_time
FROM music_titles as music, cp
WHERE music.music_price > mf_discount_price("Star Wars", cp.begin_time,
                                            cp.end_time)
AND music.begin_time <= cp.begin_time
AND cp.begin_time < music.end_time;

CREATE FUNCTION mf_discount_price(title CHARACTER(10),
                                  begin_time DATE),
                                  end_time DATE)
    RETURNS DECIMAL(5,2)
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE regular_price DECIMAL(5,2);
    SET regular_price = (SELECT movie_price
                         FROM movie_titles as movie
                         WHERE movie.movie_title = title
                         AND movie.begin_time <= begin_time
                         AND begin_time < movie.end_time);
    IF (regular_price > 10) THEN
      RETURN regular_price * 0.8;
    ELSE
      RETURN regular_price;
  END;
```

FIGURE 8.3. The example query mapping using maximally-fragmented slicing

the function. We keep it as a parameter because it is needed in the case that the function modifies data in some temporal tables. In the following example, we extend the definition of table `movie_titles` by adding one column `discount_price`. Function `discount_price_with_update()` computes the same thing as `discount_price()` but with the side effect of updating the `discount_price` column of the table `movie_titles`. The definition of the function is shown in Figure 8.4 and the resulting function from the mapping is shown in Figure 8.5. The update statement in the original function is a sequenced update applied to the input valid period. It is mapped to two insert statements and three update statements in Figure 8.5 [45] over the period of applicability `[begin_time, end_time)`.

Specifically, the two time parameters are added to each function. Inside the func-

```
CREATE FUNCTION discount_price_with_update(title CHARACTER(10))
    RETURNS DECIMAL(5,2)
    LANGUAGE SQL
    WRITE SQL DATA
  BEGIN
    DECLARE regular_price DECIMAL(5,2);
    DECLARE dp DECIMAL(5,2);
    SET regular_price = (SELECT movie_price
                         FROM movie_titles
                         WHERE movie_title = title);
    IF (regular_price > 10) THEN
      dp = regular_price * 0.8;
    ELSE
      dp = regular_price;

    UPDATE movie_titles
    SET discount_price = dp
    WHERE movie_title = title;

    RETURN dp;
  END;
```

FIGURE 8.4. Function `discount_price_with_update()`

tion, each select statement is changed to a timeslice at the passed-in begin time, and modify statements are sequenced over the passed-in period. In maximally-fragmented slicing, no control statements need to be mapped.

8.2.4 External Routines

There are SQL routines and external routines in SQL/PSM. We have discussed the mapping of SQL routines. For external routines written in other programming languages such as C/C++ and Java, the same mapping applies to the source code of the function. When the external function has SQL data manipulation statements, its source code is usually available to DBMS for precompiling. In the case that a PSM is a compiled C function, the C function does not access any database tables, thus

```
CREATE FUNCTION mf_discount_price_with_update(title CHARACTER(10))
                                              begin_time DATE),
                                              end_time DATE)
    RETURNS DECIMAL(5,2)
    LANGUAGE SQL
    WRITE SQL DATA
  BEGIN
    DECLARE regular_price DECIMAL(5,2);
    DECLARE dp DECIMAL(5,2);
    SET regular_price = (SELECT movie_price
                         FROM movie_titles
                         WHERE movie_title = title
                         AND movie.begin_time <= begin_time
                         AND begin_time < movie.end_time);

    IF (regular_price > 10) THEN
      dp = regular_price * 0.8;
    ELSE
      dp = regular_price;

    INSERT INTO movie_titles
    SELECT m.movie_id, m.movie_title, m.movie_price,
           m.discount_price, m.begin_time, begin_time
    FROM movie_titles m
    WHERE m.movie_title = title AND m.begin_time < begin_time
    AND begin_time < m.end_time;

    INSERT INTO movie_titles
    SELECT m.movie_id, m.movie_title, m.movie_price,
           m.discount_price, end_time, m.end_time
    FROM movie_titles m
    WHERE m.movie_title = title AND m.begin_time < end_time
    AND end_time < m.end_time;

    UPDATE movie_titles m SET m.discount_price = dp
    WHERE m.movie_title = title AND m.begin_time < end_time
    AND m.end_time > begin_time;

    UPDATE movie_titles m SET m.begin_time = begin_time
    WHERE m.movie_title = title AND m.begin_time < begin_time
    AND m.end_time > begin_time;

    UPDATE movie_titles m SET m.end_time = end_time
    WHERE m.movie_title = title AND m.begin_time < end_time
    AND m.end_time > end_time;

    RETURN dp;
  END;
```

FIGURE 8.5. Mapping Result of Function `discount_price_with_update()`

there is no need to map it.

8.3 Per-Statement Slicing

As in τXQuery, maximally-fragmented slicing has to evaluate a stored routine many times if the base tables change frequently over time. In order to evaluate the stored routines only once, we carry on the analogy of per-expression slicing in this section. Since the counterpart of expressions in SQL are statements, we call this method *per-statement slicing*. The idea of per-statement slicing is to map each sequenced stored routine to a semantically equivalent conventional routine that operates on temporal tables. Therefore, each SQL control statement inside the routines should also operate on temporal tables. In Section 8.3.3, we describe how to map each sequenced SQL control statement to semantically equivalent SQL statements.

There are three steps in per-statement slicing. We first convert some of the sequenced control statements into time-varying data, which we call temporal closure. To simplify the mapping of each control statement, we then normalize the SQL control statements. The intention is similar to the normalization of XQuery. In SQL there is no normalization provided by the standard organization. In this section, we will first describe the temporal closure of the control statements. Then normalization of the control statements is briefly demonstrated. The translation of the rest of the control statements will be given in Section 8.3.3.

8.3.1 Temporal Closure

We convert the control statements into data which can then be sequenced. The control flow of the program will then determined by the value of the data. To express the notion of converting via program translation a control construct in a sequenced operation into time-varying data, we use the term *temporal closure*. A conventional closure consists of a function coupled with its environment, such as local variables from its

declaration. In doing so, the function is effectively turned into data, admittedly, data that can be later evaluated. (We thank David S. Wise for this insight.) Specifically, in SQL/PSM we temporally close `LEAVE` statement and `ITERATE` statement. These two are converted to boolean variables. These boolean variables become sequenced boolean variables after the mapping.

8.3.1.1 `LEAVE` Statement

The `LEAVE` statement is the direct reason that drove us to temporally close control statements. The syntax of the `LEAVE` statement is as follows.

⟨leave statement⟩ ::= `LEAVE` ⟨statement label⟩

A ⟨statement label⟩ indicates the start position of a loop-like statement (⟨loop statement⟩, ⟨while statement⟩, ⟨repeat statement⟩, or ⟨for statement⟩) or a ⟨compound statement⟩ enclosed with `BEGIN` and `END`. A ⟨leave statement⟩ allows the program control to jump to the end of the block labeled by the ⟨statement label⟩. When used in a loop-like statement, the semantics of ⟨leave statement⟩ is the same as that of "break" in C/C++. It is hard to find the semantically equivalent SQL statements for the sequenced ⟨leave statement⟩. Therefore, we tried to remove ⟨leave statement⟩ before we map each control statement. Next, we will look at the way we remove the ⟨leave statement⟩.

Let's start with a simplest case — leaving a compound statement. This case can be represented by the following piece of code.

```
label:
BEGIN
  ⟨SQL statement list 1⟩;
  LEAVE label;
  ⟨SQL statement list 2⟩;
END label
```

This compound statement can be converted to the following semantically equivalent code without a ⟨leave statement⟩.

```
DECLARE leave_label BOOL;
SET leave_label = FALSE;
label:
BEGIN
  ⟨SQL statement list 1⟩;
  leave_label = TRUE;
  IF NOT(leave_label) THEN
  BEGIN
    ⟨SQL statement list 2⟩;
  END
END label
```

The above temporal closure can be described as follows. Declare a boolean variable for the label and initialize the variable as `FALSE`. Replace the ⟨leave statement⟩ with an assignment statement that sets the boolean variable to `TRUE`. All the statements following the ⟨leave statement⟩ are put into an ⟨if statement⟩ with the condition the value of the boolean variable is `FALSE`. Once the value of the boolean variable turns `TRUE`, the statement list that follows the ⟨leave statement⟩ in the compound statement will not be executed. After the ⟨leave statement⟩ is removed, the label is no longer used in the above code. We retain the statement only to show the boundary of the original compound statement.

Besides compound statements, a ⟨leave statement⟩ can be used in a loop-like statement. Let's use ⟨loop statement⟩ as an example to see the conversion. The temporal closure of a ⟨leave statement⟩ inside other loop-like statements is analogous. The following code shows a ⟨leave statement⟩ embedded in a ⟨loop statement⟩.

```
label:
```

```
LOOP
  ⟨SQL statement list 1⟩;
  LEAVE label;
  ⟨SQL statement list 2⟩;
END LOOP label
```

After being temporally closed, the above code is converted to the following program.

```
DECLARE leave_label BOOL;
SET leave_label = FALSE;
label:
WHILE not(leave_label) DO
BEGIN
  ⟨SQL statement list 1⟩;
  leave_label = TRUE;
  IF NOT(leave_label) THEN
  BEGIN
    ⟨SQL statement list 2⟩;
  END
END WHILE label
```

The ⟨loop statement⟩ is converted to a ⟨while statement⟩ with the entry condition the value of the boolean variable is `FALSE`. The mapping of statements inside the while loop is the same as that in a compound statement. Once the boolean variable turns to `TRUE`, the statement list that follows the ⟨leave statement⟩ will not be executed, neither does the whole while loop. A similar temporal closure can be easily used for other loop-like statements (⟨while statement⟩, ⟨repeat statement⟩, and ⟨for statement⟩). We omit the details for those statements.

8.3.1.2 `ITERATE` Statement

`ITERATE` statement is another control statement that can change the control flow of a labeled compound statement or labeled loop statement. The syntax of `ITERATE` statement is as follows.

⟨iterate statement⟩ ::= `iterate` ⟨statement label⟩

The ⟨statement label⟩ is the same as in `LEAVE` statement. An ⟨iterate statement⟩ allows the program control to jump to the beginning of the block labeled by the ⟨statement label⟩. Used in a loop-like statement, the semantics of ⟨leave statement⟩ is the same as that of "continue" in C/C++. Similar to `LEAVE` statement, it is hard to find the semantically equivalent SQL statements for the sequenced ⟨iterate statement⟩. Therefore, we tried to temporally close ⟨iterate statement⟩ before we map each control statement.

The temporal closure of ⟨iterate statement⟩ is analogous to the temporal closure of ⟨leave statement⟩. When an ⟨iterate statement⟩ is embedded in a loop-like statement, it is temporally closed in a way similar to the temporal closing of ⟨leave statement⟩. An example is as follows.

```
label:
LOOP
  ⟨SQL statement list 1⟩;
  ITERATE label;
  ⟨SQL statement list 2⟩;
END LOOP label
```

After being temporally closed, the above code is converted to the following program.

```
DECLARE iterate_label BOOL;
label:
LOOP
  SET iterate_label = FALSE;
```

```
    ⟨SQL statement list 1⟩;
    iterate_label = TRUE;
    IF NOT(iterate_label) THEN
    BEGIN
      ⟨SQL statement list 2⟩;
    END
  END LOOP label
```

The ⟨iterate statement⟩ doesn't have any impact on the next iteration of the loop. So the boolean variable `iterate_label` doesn't need to be checked as the entry condition of the loop. Once the boolean variable turns to `TRUE`, the statement list that follows the ⟨iterate statement⟩ will not be executed. The same temporal closure can be easily used for other loop-like statements (⟨while statement⟩, ⟨repeat statement⟩, and ⟨for statement⟩). We omit the details for those statements.

8.3.2 Normalization

The goal of normalizing some of the control statements is to reduce the size of the control statements set to be translated. For example, `REPEAT` statement can be converted to `WHILE` statement. This way we only need to provide the translation of `WHILE` statement. In SQL there is no normalization provided by the standard organization. The intention of our normalization is not to seek a minimal core set of the control statements, but to make our mapping simple. We choose to normalize two kinds of statements.

8.3.2.1 `REPEAT` *Statement*

As in many programming languages, A `REPEAT` statement can be rewritten using `WHILE` statement. The syntax of `REPEAT` statement is as follows.

⟨repeat statement⟩ ::=

[⟨beginning label⟩]:

`REPEAT`

⟨SQL statement list⟩

`UNTIL` ⟨search condition⟩

`END REPEAT` [⟨ending label⟩]

The repeat statement is equivalent to the following while statement.

[⟨beginning label⟩]:

`BEGIN`

⟨SQL statement list⟩

`WHILE` ⟨search condition⟩ `DO`

⟨SQL statement list⟩

`END WHILE`

`END` [⟨ending label⟩]

Therefore, a sequenced ⟨repeat statement⟩ is mapped to a sequenced ⟨while statement⟩ first, then is translated to SQL/PSM statements following the mapping rule in Section 8.3.3.

8.3.2.2 `IF` *Statement*

An `IF` statement can be converted to a `CASE` statement. The syntax of `IF` statement is as follows.

⟨if statement⟩ ::=

`IF` ⟨search condition⟩

`THEN` ⟨SQL statement list 1⟩

[`ELSE` ⟨SQL statement list 2⟩]

`ENDIF`

⟨if statement⟩ is equivalent to a ⟨searched case statement⟩ below.

`CASE`

WHEN ⟨search condition⟩

THEN ⟨SQL statement list 1⟩

[ELSE ⟨SQL statement list 2⟩]

END CASE

Therefore the mapping of an ⟨if statement⟩ is the same as the mapping of a ⟨searched case statement⟩, which we describe in Section 8.3.3.

8.3.3 Translation

After normalization, the normalized routine can be translated. We first look at the translation of the whole query that invokes the stored routine. Then we show how the statements inside the routine are mapped.

8.3.3.1 Translation of the Invoking Query and the Interface of the Routine

In per-statement slicing, each routine being invoked in a sequenced query has the sequenced semantics. Therefore the output and return values are all temporal tables. This requires the signature of the routine to be changed. Since only functions can be called in an SQL query, we describe the mapping of function definitions in this section. The mapping of procedure definitions will be covered in next section. We use the semantic function $ps\,[\![\ ]\!]\ p$ to show the mapping of per-statement slicing. p is an input parameter of the semantic function indicating the period of validity of the return data of the input query.

Each sequenced function is evaluated for a particular temporal period and the return value of the sequenced function is a temporal table. Therefore, a temporal period is added to the input parameter list. The return value is a sequence of return values, each associated with a valid-time period. The formal mapping of a function definition is as follows.

$ps\,[\![\langle\text{SQL-invoked function}\rangle]\!]\ p = ps\,[\![\langle\text{SQL-invoked function}\rangle]\!]$

In this mapping, p is removed. This mapping indicates that when the function is defined, there is no valid time period applied to the definition. Each ⟨SQL-invoked function⟩ includes a ⟨function specification⟩ and a ⟨routine body⟩. We only look at the function specification in this section, since the major components of ⟨routine body⟩ are control statements which will be covered in next section. ⟨function specification⟩ defines the signature of the function including three non-terminals, namely ⟨routine name⟩, ⟨parameter declaration list⟩, and ⟨returns clause⟩. The specific mappings are shown as follows. A suffix `ps_` is added to each routine name to differentiate the sequenced function from the original function with current semantics.

ps ⟦⟨routine name⟩⟧ = `ps_`⟨routine name⟩

As in maximally-fragmented slicing, two more input parameters are added to the parameter list to indicate the valid time period the function is evaluated for.

ps ⟦⟨parameter declaration list⟩⟧ =

⟨parameter declaration list⟩, `begin_time DATE`, `end_time DATE`

The ⟨returns clause⟩ has the following syntax.

⟨returns clause⟩ ::= `RETURNS` ⟨data type⟩

The data type of the return value is mapped to a temporal table derived by a *collection type*. A collection type is a set of rows that have the same data structure. In this mapping, each row has three columns representing the return value with its valid time period.

ps ⟦⟨return clause⟩⟧ =

`RETURNS ROW(value` ⟨data type⟩, `begin_time DATE`, `end_time DATE) ARRAY`

The returned value is always a temporal table. This returned temporal table then joins with other temporal tables in the invoking query. We can follow the definition of temporal join and temporal cartesian product [47] to map the invoking temporal query to the semantically equivalent regular SQL query. If the function appears in the WHERE clause and the return value is compared with some column of other temporal tables, a temporal join will be done between the returned temporal table

and other temporal tables. Otherwise, a temporal cartesian product will be done.

As an example, let's look at the query Q2 used to illustrate maximally-fragmented slicing. Assume the sequenced function `ps_discount_price()` returns a temporal table, query Q2 is translated to the following SQL Query. The mapping from `discount_price()` to `ps_discount_price()` will be given when we describe the translation of control statement.

```
SELECT music_title,
       last_instant(music.begin_time, ps_dp.begin_time),
       first_instant(music.end_time, ps_dp.end_time)
FROM music_titles as music,
     ps_discount_price("Star Wars", MIN_TIME, MAX_TIME) as ps_dp
WHERE music.music_price > ps_dp.value
AND last_instant(music.begin_time, ps_dp.begin_time)
     < first_instant(music.end_time, ps_dp.end_time);
```

The function `ps_discount_price()` has two more arguments than then function `discount_price()`. The time period [`MIN_TIME`, `MAX_TIME`) is the query period defined by the sequenced query. The function `ps_discount_price()` returns a temporal table, which is then joined with the table `music_titles`. The function `first_instant()` [45] returns the earlier one of the two input instants. Similarly, the function `last_instant()` returns the later one of the two input instants. The additional predicates in the where clause ensure that the valid time of music price and the valid time of discount price overlap.

8.3.3.2 Translation of Control Statements

This section focuses on mapping control statements, i.e., translating sequenced SQL control statements to semantically equivalent SQL. Each scalar variable is represented by a temporal table that includes three columns — the variable itself, begin time,

and end time. Any return value of a stored function is also represented by a temporal table.

SQL/PSM control statements include the twelve types of statements listed in the following production.

⟨SQL control statement⟩ ::=

⟨call statement⟩
| ⟨return statement⟩
| ⟨assignment statement⟩
| ⟨compound statement⟩
| ⟨case statement⟩
| ⟨if statement⟩
| ⟨iterate statement⟩
| ⟨leave statement⟩
| ⟨loop statement⟩
| ⟨while statement⟩
| ⟨repeat statement⟩
| ⟨for statement⟩

We classify the twelve kinds of statements into five categories.

- **Temporally closed statements**. ⟨leave statement⟩ and ⟨iterate statement⟩. They are removed in the temporal closing stage.

- **Standalone statements**. These include ⟨call statement⟩, ⟨return statement⟩, ⟨assignment statement⟩, ⟨compound statement⟩, and ⟨loop statement⟩. Each of them has a different mapping definition from others.

- **Condition statements**. These include ⟨case statement⟩ and ⟨if statement⟩, which usually need to use a cursor in the mapping. ⟨if statement⟩ is removed in the normalization stage.

- **Conditional loop statements**. These are the loop-like statements with conditions. ⟨while statement⟩, ⟨repeat statement⟩, and ⟨for statement⟩ fall in this category. They are combination of loop-like statements and condition statements. Among them, ⟨repeat statement⟩ is removed in the normalization stage.

We examine each kind of control statements in turn except for those statements that have been removed in the first two stages. We start with standalone statements.

1. ⟨call statement⟩ ::= `CALL` ⟨routine invocation⟩

   $ps\ [\![\langle\text{call statement}\rangle]\!]\ p =$ `CALL` $ps\ [\![\langle\text{routine invocation}\rangle]\!]\ p$

   The routine invoked in a ⟨call statement⟩ is a procedure. The only difference between procedures and functions is the procedure can have more than one output parameter. We define each output parameter as a collection that represents a temporal table.

2. ⟨return statement⟩ ::= `RETURN` ⟨value expression⟩

   Each ⟨return statement⟩ is mapped to an INSERT statement that inserts some tuples into the temporal table that stores all the return values. At the end of the function, one ⟨return statement⟩ is added to return the temporal table. The invoking query will then get the return value and use it as a temporal table.

   $ps\ [\![\langle\text{return statement}\rangle]\!]\ p =$

   `INSERT INTO TABLE ps_return_tb`

   $ps\ [\![\langle\text{value expression}\rangle]\!]\ p$

   The sequenced ⟨value expression⟩ returns a temporal table that has three columns: one value with the same type of the ⟨value expression⟩, one begin_time, and one end_time of the valid time period of the value. ⟨value expression⟩ could be a literal, a variable, a select statement that returns a single value, or a function that returns a single value. It is trivial to map a literal into a temporal tuple.

We just need to add the valid period for the literal. A variable is mapped to a select statement that retrieves the tuples from the temporal table (the sequenced variable). The mapping of the sequenced select statement is given in previous research [47], and the mapping of a sequenced function call is defined in Section 8.3.3.

3. ⟨assignment statement⟩ ::= `SET` ⟨assignment target⟩ = ⟨value expression⟩

   The ⟨assignment target⟩ is usually a variable. A variable inside a routine is mapped to a temporal table. Therefore a sequenced assignment statement tries to insert tuples into or update the temporal table for a certain period. Intuitively the assignment statement should be mapped to a sequenced insert or update. Here we map it to a sequenced delete followed by an insert to the target temporal table. If there are tuples valid in the input time period, they are first deleted, then new tuples are inserted. It is the same as sequenced update. If there are no tuples valid in the input time period, a new tuple is inserted. The inserted tuples are returned from the sequenced ⟨value expression⟩.

   ps ⟦⟨assignment statement⟩⟧ *p* =
   ps ⟦`DELETE FROM TABLE` ⟨assignment target⟩⟧ *p*;
   `INSERT INTO TABLE` ⟨assignment target⟩
   ps ⟦⟨value expression⟩⟧ *p*

   The mapping of a sequenced deletion to semantically equivalent SQL deletion have been addressed in previous research [45]. Since it is not the focus of this dissertation, we will not show the details of the mapping. However, an example mapping will be shown at the end of this section.

4. ⟨compound statement⟩ ::=
   [⟨beginning label⟩:]
   `BEGIN`

[⟨local declaration list⟩]

[⟨local cursor declaration list⟩]

[⟨local handler declaration list⟩]

[⟨SQL statement list⟩]

`END` [⟨ending label⟩]

$ps\ [\![$⟨compound statement⟩$]\!]\ p =$

[⟨beginning label⟩`:`]

`BEGIN`

$[ps\ [\![$⟨local declaration list⟩$]\!]\ p]$

$[ps\ [\![$⟨local cursor declaration list⟩$]\!]\ p]$

$[ps\ [\![$⟨local handler declaration list⟩$]\!]\ p]$

$[ps\ [\![$⟨SQL statement list⟩$]\!]\ p]$

`END` [⟨ending label⟩]

⟨local declaration list⟩ ::= ⟨local declaration⟩ (`;` ⟨local declaration⟩)

⟨local declaration⟩ ::= ⟨SQL variable declaration⟩ | ⟨condition declaration⟩

⟨SQL varible declaration⟩ ::= `DECLARE` ⟨SQL variable name⟩ ⟨data type⟩

$ps\ [\![$⟨SQL variable declaration⟩$]\!]\ p =$

`DECLARE LOCAL TEMPORARY TABLE` ⟨SQL variable name⟩

`(value` ⟨data type⟩`,`

`begin_time TIME,`

`end_time TIME);`

An SQL variable declaration is mapped to the declaration of a temporary temporal table that stores the sequenced values of the variable. The table has three columns: the value with the same type of the original variable, the begin_time, and the end_time of the valid time period of the value. Any reference to the value of the variable will be mapped to a reference to the value column

of the table. Any assignment to the variable will be mapped to a sequenced deletion followed by an insertion to the table. There are no changes to the ⟨condition declaration⟩.

⟨local cursor declaration list⟩ ::= ⟨declare cursor⟩ (; ⟨declare cursor⟩)

⟨declare cursor⟩ ::=

`DECLARE` ⟨cursor name⟩ `CURSOR`

`FOR` ⟨query expression⟩

In a sequenced cursor declaration, only the last non-terminal (the ⟨query expression⟩) needs to be mapped.

ps ⟦⟨declare cursor⟩⟧ *p* =

`DECLARE` ⟨cursor name⟩ `CURSOR`

`FOR` *ps* ⟦⟨query expression⟩⟧ *p*

⟨local handler declaration list⟩ ::= ⟨handler declaration⟩ (; ⟨handler declaration⟩)

⟨handler declaration⟩ ::=

`DECLARE` ⟨handler type⟩ `HANDLER`

`FOR` ⟨condition value list⟩

⟨SQL procedure statement⟩

Similarly, in a sequenced handler declaration, only the ⟨SQL statement⟩ needs to be mapped.

ps ⟦⟨handler declaration⟩⟧ *p* =

`DECLARE` ⟨handler type⟩ `HANDLER`

`FOR` ⟨condition value list⟩

ps ⟦⟨SQL statement⟩⟧ *p*

5. ⟨loop statement⟩ ::=

[⟨beginning label⟩ :]

`LOOP`

⟨SQL statement list⟩

END LOOP [⟨ending label⟩]

In a sequenced ⟨loop statement⟩, only the ⟨SQL statement list⟩ inside the loop needs to be mapped to the sequenced semantics.

ps ⟦⟨loop statement⟩⟧ p =

[⟨beginning label⟩:]

LOOP

ps ⟦⟨SQL statement list⟩⟧ p

END LOOP [⟨ending label⟩]

6. From now on, we look at the condition statements. The condition in a ⟨case statement⟩ or an ⟨if statement⟩ is usually a boolean expression. We map the sequenced boolean expression into a temporal table since the boolean value could change during the input valid time period. We then declare a cursor on this temporal table. The control flow will be determined by the value of the boolean column in the associated valid-time period.

⟨case statement⟩ ::= ⟨simple case statement⟩ | ⟨searched case statement⟩

⟨simple case statement⟩ ::=

CASE ⟨value expression 1⟩

WHEN ⟨value expression 2⟩

THEN ⟨SQL statement list 1⟩

[ELSE ⟨SQL statement list 2⟩]

END CASE

The semantics of the ⟨simple case statement⟩ is as follows. ⟨value expression 1⟩ is evaluated first. Then ⟨value expression 2⟩ is evaluated. If the values are equal, execute ⟨SQL statement list 1⟩. Otherwise, execute ⟨SQL statement list 2⟩ if there is an ELSE clause.

In sequenced ⟨simple case statement⟩, each value expression returns a temporal table. So we declare a cursor for the boolean expression"⟨value expression 1⟩= ⟨value expression 2⟩". For each value of this boolean expression, the case statement is executed. (In fact, in the following mapping the if statement is used since it carries the same semantics).

ps ⟦⟨simple case statement⟩⟧ *p* =

```
DECLARE ps_case_cursor CURSOR
FOR (SELECT
     ve1.value = ve2.value,
     last_instant(ve1.begin_time, ve2.begin_time),
     first_instant(ve1.end_time, ve2.end_time)
     FROM   ps ⟦⟨value expression 1⟩⟧ p AS ve1,
            ps ⟦⟨value expression 2⟩⟧ p AS ve2,
     WHERE last_instant(ve1.begin_time, be2.begin_time)
         < first_instant(ve1.end_time, ve2.end_time))
DECLARE ps_case_not_found CONDITION FOR SQLSTATE '02000';
OPEN ps_case_cursor;
FETCH ps_case_cursor INTO ps_case_result;
WHILE NOT(ps_case_not_found) DO
  IF (ps_case_result.value)
  THEN ps ⟦⟨SQL statement list 1⟩⟧ ps_case_result.validtime;
  [ELSE ps ⟦⟨SQL statement list 2⟩⟧ ps_case_result.validtime;]
  FETCH ps_case_cursor INTO ps_case_result;
END WHILE
```

⟨searched case statement⟩ ::=

```
CASE
  WHEN ⟨search condition⟩
```

THEN ⟨SQL statement list 1⟩

[ELSE ⟨SQL statement list 2⟩]

END CASE

⟨search condition⟩ ::= ⟨boolean value expression⟩

Since a ⟨search condition⟩ is a ⟨boolean value expression⟩, the mapping of searched case statements is a generalization to the mapping of simple case statement. Here the boolean expression is not necessarily in the format of "A=B". It could be inequality comparison, or negation, or conjunction of boolean expressions, etc. A cursor needs to be declared for the sequenced ⟨search condition⟩, i.e., ⟨boolean value expression⟩. The rest of the mapping is the same as in ⟨simple case statement⟩.

7. The last category is the conditional loop statements. These are loop-like statements with condition indicating when to exiting or executing the loop. As in condition statements, the condition is converted to a temporal table and a cursor is declared on this temporal table. At each iteration of the loop, the cursor value is fetched to determine the control flow.

   ⟨while statement⟩ ::=

   [⟨beginning label⟩:]

   WHILE ⟨search condition⟩ DO

   ⟨SQL statement list⟩

   END WHILE [⟨ending label⟩]

   The search condition is a boolean type value expression. As in ⟨case statement⟩, we declare a cursor for the boolean value expression. For each returned tuple, the while statement is called once on the valid period of the tuple. At the end of each iteration of the while statement, add one select statement to reevaluate the search condition and assign the result to the variable that will be checked

by the while statement.

ps ⟦⟨while statement⟩⟧ *p* =

```
DECLARE ps_while_cursor CURSOR
FOR ps ⟦⟨value expression⟩⟧ p
DECLARE NOT_FOUND CONDITION FOR SQLSTATE '02000';
OPEN ps_while_cursor;
FETCH ps_while_cursor INTO ps_while_result;
WHILE NOT(NOT_FOUND) DO
  [⟨beginning label⟩:]
  WHILE (ps_while_result.value) DO
    ps ⟦⟨SQL statement list⟩⟧ ps_while_result.validtime
    ps_while_result = ps ⟦⟨value expression⟩⟧ ps_while_result.validtime
  END WHILE [⟨ending label⟩]
  FETCH ps_while_cursor INTO ps_while_result;
END WHILE
```

8. ⟨for statement⟩ ::=

   [⟨beginning label⟩]:
   FOR ⟨variable name⟩AS
    [⟨cursor name⟩ CURSOR FOR]⟨cursor specification⟩
    DO ⟨SQL statement list⟩
   END FOR [⟨ending label⟩]

   In ⟨for statement⟩, the ⟨cursor specification⟩ and the ⟨SQL statement list⟩ need to be mapped to the sequenced semantics. A ⟨cursor specification⟩ is a query expression, which we have already know how to map.

   ps ⟦⟨for statement⟩⟧ *p* =

   [⟨beginning label⟩]:
   FOR ⟨variable name⟩AS

[⟨cursor name⟩ `CURSOR FOR`]

ps ⟦⟨cursor specification⟩⟧ *p*

`DO`

ps ⟦⟨SQL statement list⟩⟧ ⟨variable name⟩`.validtime`

`END FOR` [⟨ending label⟩]

8.3.4 An Example

Now we apply the per-statement slicing mapping rules to the example function in Figure 7.2. We repeat the original function here for convenience.

```
-------------------------
-- The original function
-------------------------
CREATE FUNCTION discount_price(title CHARACTER(10))
    RETURNS DECIMAL(5,2)
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE regular_price DECIMAL(5,2);
    SET regular_price = (SELECT movie_price
                         FROM movie_titles
                         WHERE movie_title = title);
    IF (regular_price > 10) THEN
      RETURN regular_price * 0.8;
    ELSE
      RETURN regular_price;
  END;

-----------------------------------------------------------
-- The mapping result
-- Add two more parameters to indicate the valid period
-- Return value is an array
-----------------------------------------------------------
CREATE FUNCTION ps_discount_price(title CHARACTER(10),
                                  begin_time DATE, end_time DATE)
    RETURNS ROW(value DECIMAL(5,2), begin_time DATE, end_time DATE)
            ARRAY
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    -- Declare a temporal table for the return value
    DECLARE ps_return_tb ROW(value DECIMAL(5,2), begin_time DATE,
                             end_time DATE) ARRAY;
```

```
  -- Map the variable regular_price to a temporal table
  DECLARE LOCAL TEMPORARY TABLE regular_price
    (value DECIMAL(5,2), begin_time DATE, end_time DATE);

  -- Map the assignment statement to an insertion to the table
  INSERT INTO TABLE regular_price
  SELECT movie_price,
         last_instant(movie.begin_time, begin_time),
         first_instant(movie.end_time, end_time)
  FROM movie_titles as movie
  WHERE movie.movie_title = title
  AND last_instant(movie.begin_time, begin_time)
      < first_instant(movie.end_time, end_time);

  -- Declare a cursor for the boolean expression in the IF statement
  DECLARE ps_if_cursor CURSOR
  FOR (SELECT ve1.value > 10,
              last_instant(ve1.begin_time, begin_time),
              first_instant(ve1.end_time, end_time)
       FROM regular_price AS ve1
       WHERE last_instant(movie.begin_time, begin_time)
             < first_instant(movie.end_time, end_time);
  DECLARE ps_if_not_found CONDITION FOR SQLSTATE '02000';
  OPEN ps_if_cursor;
  FETCH ps_if_cursor INTO ps_if_result;
  WHILE NOT(ps_if_not_found) DO
    IF (ps_if_result.value)
    THEN INSERT INTO ps_return_tb
         VALUES (ps_if_result.value * 0.8,
                 ps_if_result.begin_time,
                 ps_if_result.end_time);
    ELSE INSERT INTO ps_return_tb
         VALUES ps_if_result;
    FETCH ps_if_cursor INTO ps_if_result;
  END WHILE;
  RETURN ps_return_table;
END;
```

8.4 Comparison

So far, we have proposed two different ways to map sequenced SQL/PSM to semantically equivalent regular SQL/PSM. Maximally-fragmented slicing applies small, isolated changes to the routines by adding simple predicates to the SQL statements inside the routines. However, in this approach, we need to call the routines as many

times as the number of constant periods in the input period. Per-statement slicing produces complex routines by mapping every statement inside the routines. The resulting sequenced routine becomes much longer than the original one. But each routine is only called once for each input value.

Similar to the different approaches for τXQuery, there are queries and data that favor each approach. If a sequenced query asks for the result in a very short valid time period, maximally-fragmented slicing should perform better because it has the less complex statements in the routine and a few calls to each routine. On the other hand, if a sequenced query requires the result for a very long valid-time period and the data changes frequently in this period, the number of calls to the routine could be large for the maximally-fragment slicing. In this case, per-statement slicing will outperform maximally-fragmented slicing.

The next chapter empirically evaluates this tradeoff.

Chapter 9
Performance Study for Temporal SQL/PSM

There are two different time-slicing techniques presented in the last chapter. We would like to find out which one is more efficient by running performance experiments. As we experienced in the performance study for temporal XML, there are no benchmarks even for SQL/PSM, let alone temporal SQL/PSM. This time we transformed and extended the XBench to effect a temporal SQL/PSM benchmark. Section 9.1 provides more details about how it was extended. The experimental setup and the results are discussed from Section 9.2 to Section 9.6. Finally, Section 9.7 summarizes the conclusions of the performance study.

9.1 Transform the Benchmark

Since XBench was designed to study XQuery on XML data, we need to map the XML data to relational data and map the XQuery queries to SQL/PSM queries. We extended XBench in the following way. First the DC/SD XML schema was mappped to a relational schema; then valid time periods were added to the relational data by simulating the evolving of the data; next the queries were transformed to SQL/PSM queries; and finally, temporal keywords were added to achieve a temporal SQL/PSM benchmark.

The XML schema of the DC/SD XML document was mapped to five relational tables: `items`, `authors`, `item_authors`, `publishers`, and `related_items`. A non-temporal XML document of 10GB was generated by the generator of XBench. The document is a catalog of books each of which is represented by an `item` element.

We shredded one tenth of the `item` elements (about 1GB) to the five relational tables according to the relational schema. This resulted in the initial snapshot of the database. The rest of the `item` elements are also shredded but stored separately as the repository in the simulation step. *Shredding* XML data is the process of converting the hierarchical XML data into the two dimensional table. The schema of the five temporal tables are shown in Appendix L.

Assume the starting time of the temporal snapshot was st. Every tuple in the initial snapshot was associated with the valid time period $[st, forever)$. The simulation program set the current time as st, then it generated one interval i based on the predefined update frequency (the distribution of the valid period length of each snapshot will be specified in Section 9.2). The program advanced the clock by i. The current time was set as $st + i$. At each time point, the program randomly chose a number of tuples from the current snapshot and randomly chose an operation (insert, delete, or update) for each of the chosen tuple. For the update operation, the program randomly chose some time-varying columns to change. For example, at some time point, the program decided to delete three `items` tuples, update two `authors` tuples on the `phone_number` and `email_address` columns, respectively, and insert one `publishers` tuple. A delete operation set the ending time of the tuple to the current time; an insert operation inserted a new tuple with the valid period $[current\_time, forever)$; and an update operation was a delete of the current tuple with an insertion of a new tuple. Both insertion and update needed new data which came from the repository obtained from the XBench data generator. Once some data has been added to the temporal database, it was removed from the repository. The program advanced the clock until the required number of changes have been generated. In this way, we obtained a temporal relational database.

The queries in XBench are written in XQuery. We transformed the queries to SQL queries with functions and procedures embedded. Not all the queries in XBench workload can be translated to SQL with PSM. We removed seven queries but added

one query to the workload. Q4, Q5, and Q12 were removed because they relied on the sequence order of XML documents which are not applicable to relational data. Q17 was removed since it tested on the text search in XML documents. Q1, Q8, and Q19 were removed since they are too simple to be queried with PSM and the rest of the queries had PSMs contain all the control statements. We duplicate Q3 by writing it in different control statements. Thus, we ended up with a nontemporal workload with ten queries. We then added the keyword `validtime` and an optional period $[st, et]$ in front of each query. How the period is decided is explained in Section 9.2. All the queries were extended to sequenced queries. The non-temporal SQL queries with PSMs are shown in Appendix M.

We ran the sequenced queries against the temporal relational database. The experimental setup is discussed in Section 9.2. The experiments examine the performance of different time-slicing techniques. Instead of implementing the stratum, we manually mapped the sequenced queries to semantically equivalent SQL/PSM queries by using different time-slicing techniques, which were then evaluated on the Oracle 10g engine. Sections 9.3 to 9.6 demonstrate the results of running the queries over periods of different lengths.

9.2 Experimental Setup

All experiments were conducted on a 1.86GHz Intel Core 2 machine with 2GB of RAM and one 300GB MAXTOR STM3320620A disk drive running Windows XP. We chose Oracle 10g as the underlying SQL engine. The reason is that Oracle is one of the main-stream DBMS used in today's database management and its PL/SQL supports most of the functions in standard SQL/PSM.

The initial parameters used when the temporal data was generated are as follows. The starting time of the relational database is `2002-01-01`. The length of the valid period of each snapshot is uniformly distributed between 1 to 30 days. The database

stores the catalog history from `2002-01-01` to now. The total number of change points in this database is 125 and the last change time is `2007-12-30`. At each change point, the number of records changed is uniformly distributed between 1 and 5. The size of the temporal database is 1.01GB. Another database setting will be introduced in Section 9.6.

For all the database settings, we set the database buffer to be 256 MB, which is about one quarter of the database size. We run the queries with a cold cache by reading a big file into the memory before running each query.

9.3 Querying over the Whole Timeline

In the first experiment, we ran all the sequenced queries over the whole timeline. To do this, we add the keyword `validtime` before each query without the optional period. The query should return all the data in the history that satisfies the query predicates. For a given temporal database and query, this is the worst-case performance for both time-slicing techniques because in general, more data must be accessed when the query period is long. The elapsed time of different queries using different techniques are grouped by queries in Figure 9.1. The right-most group is the average elapsed time of different techniques over all the queries.

The results of all the queries are shown in Figure 9.1. In Figure 9.1, MAX (maximally-fragmented slicing) performed worse than PER_ST (per statement slicing) for all the queries since it slices the whole database at all the change points. As mentioned in Section 9.2, this database has 128 change points. Therefore, MAX sliced the database 128 times and executed the non-temporal queries 128 times against the snapshot of the database at each change point. PER_ST doesn't need to slice the whole database at every change point. Instead, it only needs to slice the intermediate results inside the PSMs. An intermediate result is usually the temporal table for a variable or a record resulted from a temporal query and will be manipulated by some

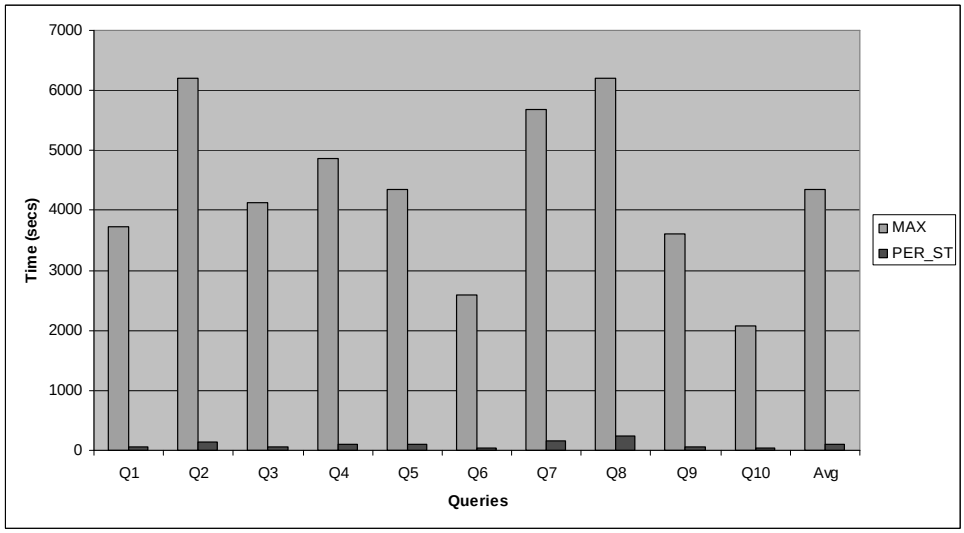


FIGURE 9.1. Query over the whole timeline

control statements. Therefore, they are much smaller than the whole database and have much less change points. Thus we see a big performance difference (about 43 times in average) between the two time-slicing techniques.

Detailed results show that the queries are CPU-bound in both slicing methods. The first reason is we created indexes on the tables to reduce the I/O. The second reason is that we make the queries complex after the mapping. The optimizer spends more CPU time to optimize the sequenced queries than the non-temporal queries. When mapping the queries, we add new predicates to the timestamps in both slicing methods. Timestamps are also added to the parameter list of the functions. For sequenced functions, timestamps are added to the return value as well. In per-statement slicing, each control statement is mapped to a few lines of control statements (about three to ten times longer). These mostly increase the CPU time of the query processing but have very small impact on I/O.

On the other hand, PER_ST makes the SQL query more complex than the non-temporal queries by adding time period and additional non-equality predicates to the queries. We expected that MAX performed better when the input time period of the

queries is short enough. The experiment for short input time period is demonstrated in the next section.

9.4 Querying over a Short Period

In this experiment, we ran all the sequenced queries over a short period, that of one day. We placed `validtime [2003-01-01, 2003-01-02]` in front of each query, to see the effect of the length of query period. The results are shown in Figure 9.2.

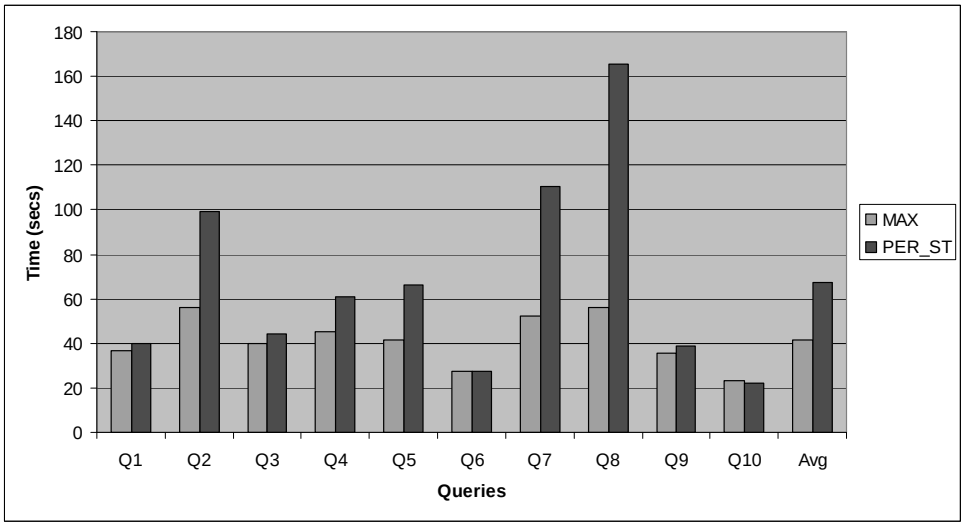


FIGURE 9.2. Querying on a short period

MAX had dramatic improvement of its performance as compared with the performance in Section 9.3. The reason is that it needs to slice the database only once in this experiment, while in the previous experiment, it sliced the database 128 times. In the short period experiment, PER_ST also had slightly better performance than it was in the experiment for long period. However it was not as good as MAX in this experiment. PER_ST did not take advantage of short query period too much because it didn't have many time-slicing even in the previous experiment. In addition, its code complexity became dominating so that it took more time to run than the simple

code in MAX.

9.5 Querying over Different Time Periods

The two extreme cases were examined in the last two sections. How about the cases in between? Our expectation was that it depends on the temporal feature of the database. When the input time period for queries is very long, the number of change points is very big. MAX cannot perform well. In our temporal database setting, since the number of tuples changed at each change point is very small compared to the size of the database, the intermediate results that PER_ST sliced is of small sizes. PER_ST won on the much less slicing. But if we shrink the length of the input time period, the number of change points will decrease. The performance of MAX will improve steadily while the performance of PER_ST won't benefit much. When the length of the input time period becomes short enough, the performance of MAX will be better than that of PER_ST, which has more complex code than MAX.

To verify the hypothesis, We ran the queries with the input period length varying from one day, one month, six months, one year, to the whole timeline. The result is shown in Figure 9.3.

Figure 9.3 shows the average query time for the ten queries when the length of the query periods vary. When querying over an extremely short time period, MAX had better performance than PER_ST. the performance of MAX degraded while the length of the query periods increased. The performance of PER_ST went down slightly. When the length of the query periods was between one month and six months, PER_ST outperformed MAX and kept much better performance than MAX for even longer length of query periods. This was exactly the same as we expected.

Which time-slicing technique is better depends on the temporal feature of the database. So far we used the database that doesn't change very often. The amount of data changed is relatively small at each change point as well as in the whole

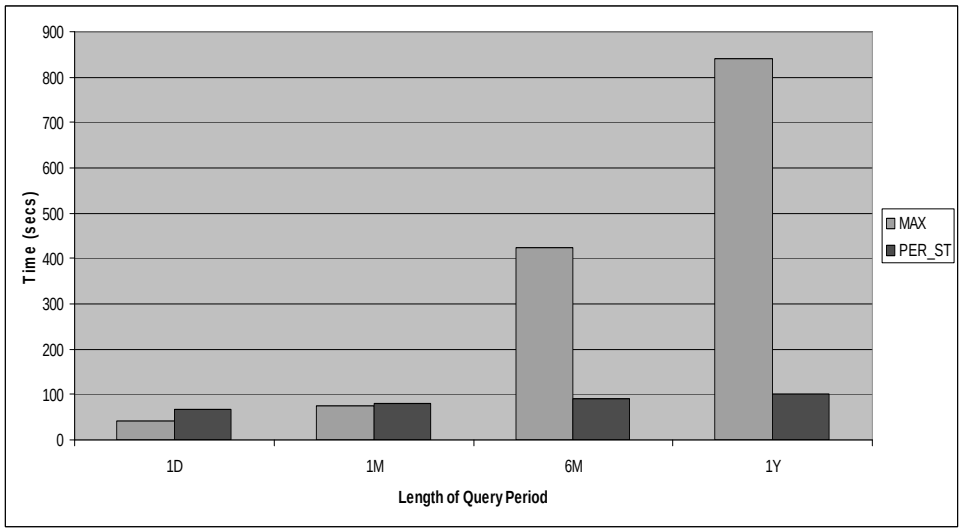


FIGURE 9.3. Varying the Length of Query periods

timeline. We now examine the performance of the two techniques in a more dynamic database.

9.6 Querying against a Frequently Changed Database

In this experiment, we changed the database setting. The length of the valid period of each snapshot is uniformly distributed between 1 to 5 days. The database stores the catalog history from `2002-01-01` to now. The total number of change points in this database is 649 and the last change time is `2007-12-30`. At each change point, the number of records changed is uniformly distributed between 20 and 40. The size of the temporal database is 1.5GB. Compared to the database used in the previous sections, this database changed 5 times more frequently and at change point, 10 times more data were changed. The size of the database is bigger since any changes in the database result in more tuples. The history is never removed but accumulated. The parameters we used to generate the temporal data are by no means the changing frequencies of a typical temporal database. Actually, we are not aware of any typical

changing frequency of temporal databases. The changing frequency totally depends on the application. Our objective here is to generate data with different changing frequency to see the relative behavior of the two mapping methods.

The workload ran over all the different time periods used in the previous experiments. The results are shown in Figure 9.4.

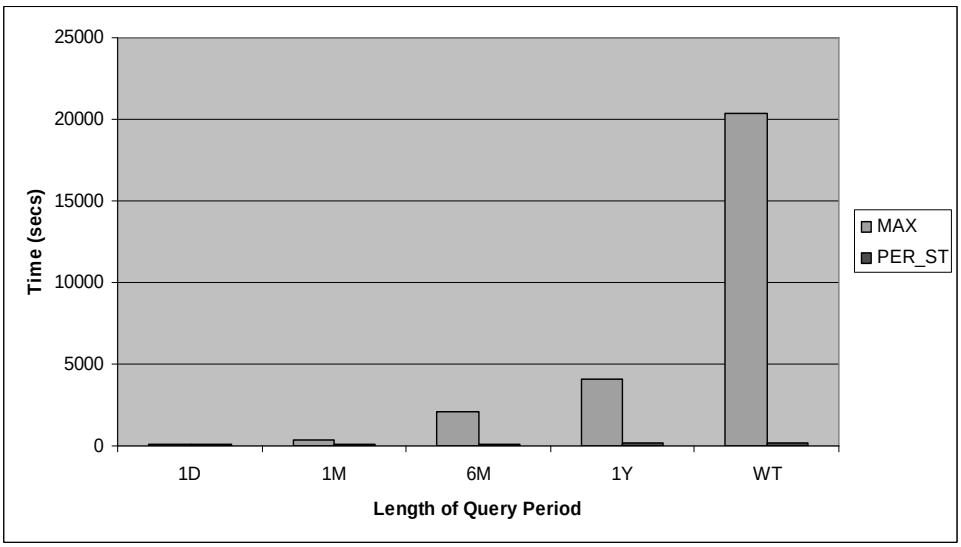


FIGURE 9.4. Querying against a Frequently Changed Database

When querying over the whole timeline, the difference between MAX and PER_ST is larger than in previous experiments. The reason is MAX had to slice the whole database at five times more change points. This highest bar of MAX in the graph made it difficult to see the performance for other lengths of query period. We removed the last group of bars for the whole timeline from Figure 9.4 and show the rest of them in Figure 9.5.

In the more dynamic database, the performance trend of MAX and PER_ST remained the same as in the previous database setting. But since data were changed more frequently, the same query period contained more change points, thus MAX needed to slice the database more times. This is the reason that the performance of

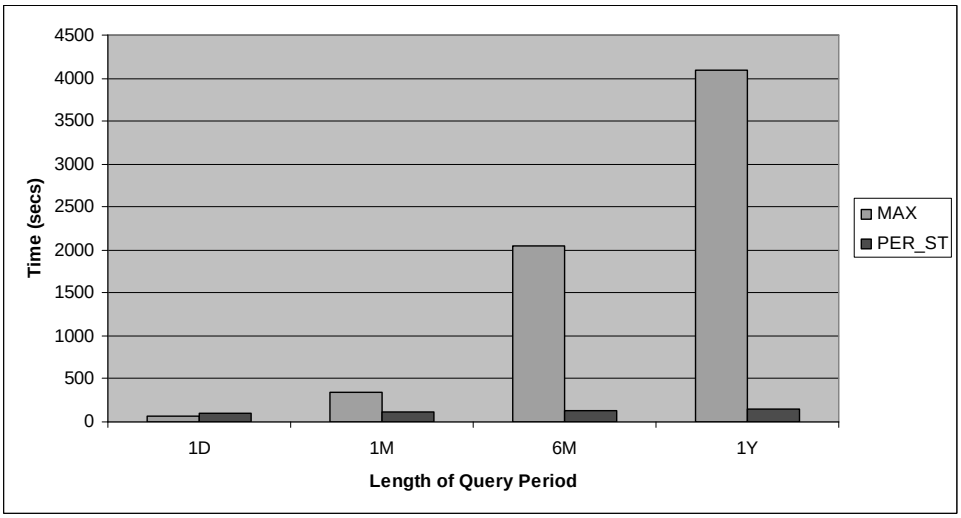


FIGURE 9.5. Querying against a Frequently Changed Database (WT removed)

MAX degraded faster in this experiment. It was already worse than PER_ST when the query period increased to 1 month.

9.7 Summary

The empirical observations show that the time-slicing techniques are applicable to all the queries in a sequenced SQL/PSM workload.

The length of the query period has a significant impact on the performance of MAX which slices the data at every change point of the database. The shorter the query period, the fewer the slices, therefore, the better the performance of MAX. It could be better than PER_ST when the query period is short enough. However, when the query period is long enough, the performance of MAX drops dramatically.

The temporal feature of the database also play an important role on the performance of MAX. The more frequently changed database has more change points in a given query period. In this case, MAX doesn't perform as good as in a less frequently changed database.

The two factors discussed above have slight impact on PER_ST because PER_ST only slices on the small intermediate results inside PSMs. The length of the query period and the dynamic feature of database both have impact on the size of the intermediate results, but it is not significant. Therefore, the performance of PER_ST doesn't degrade dramatically when the length of the query period increases or the database changing frequency increases.

In most cases PER_ST is better than MAX. But the code of PER_ST is more complex than MAX. When the query period is extremely short or the data doesn't change frequently, thus the query period doesn't contain a lot of change points, PER_ST is not as good as MAX.

Chapter 10
Conclusion and Future Work

The pervasiveness of temporal data in both relational databases and XML documents calls for effective and efficient temporal query ability. Expressing queries on time-varying (relational or XML) data by using standard query language (SQL or XQuery) is more difficult than writing queries on nontemporal data. In this dissertation, we presented minimal valid-time extensions to XQuery and SQL/PSM, focusing on the procedural aspect of the two query languages and efficient evaluation of sequenced queries.

For XQuery, we added valid time support to it by minimally extending the syntax and semantics of XQuery. We adopt a stratum approach which maps a τXQuery query to a conventional XQuery. The first part of the dissertation 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 proposed five 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 reduced the number of constant periods over which the query was evaluated. We also extended a conventional XML query benchmark to effect a temporal XML query benchmark. Experiments on this benchmark showed that in-place slicing was the best.

We then applied the same minimal extension to SQL/PSM to obtain the Temporal SQL/PSM. The stratum architecture worked for implementing Temporal SQL/PSM.

To map Temporal SQL/PSM to semantically equivalent SQL/PSM, we proposed two time-slicing techniques: maximally-fragmented slicing and per-statement slicing. For the empirical study of the performance of the two techniques, we converted XBench including both the data model and the queries to effect a temporal SQL/PSM workload. In most cases, per-statement slicing was much better than maximally-fragmented slicing.

This dissertation used the same approach to successfully apply temporal ordination to the two different query languages based on different data models. Since the two languages are not only query languages but also programming languages, we expect that the approaches proposed in this dissertation work for temporal ordination of other programming languages. However it needs to be verified by further study. It would be an interesting topic for future work. The recent XQuery standard has the ability to update XML data, which was not discussed by this dissertation. Although we expect that the approaches discussed in this dissertation work for the update, temporal ordinating of the update aspect of XQuery is also worth being researched as future work. The current mapping for τXQuery doesn't use any schema information. In the future, we can consider to utilize schema information to further improve the mapping and compare the performance of the best automatically translated queries and the well-written temporal queries in XQuery. We would like to automate other aspects of manually-written XQuery equivalents, and also study how close the existing automatic approaches get to a manually-written XQuery program. This should be studied for temporal SQL/PSM, too. The performance result can be sensitive to the parameters used to generate the changing frequency of the temporal XML and temporal relational data. In the future, a study is needed on the sensitivity of the results to the parameters.

In general if statistics of data is available, it can be used by the query optimizer to estimate the cost of different query methods. Similarly, it can also be used in the future to build a cost model for the stratum of Temporal PSM and τXQuery. This

would allow the stratum to compare the estimated cost of different slicing methods and correctly choose between the two slicing methods.

In some systems, XML data are shredded into relational database. While the user interface allows the users to use XQuery to retrieve the data, the system actually maps the XQuery expressions to SQL query since the data are stored in a relational database. In this case, how to map τXQueryto SQL is a good topic for future research.

Appendix 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>
```

Appendix 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>
```

Appendix C
Auxiliary Functions

Here we provide an implementation of all of the auxiliary functions used in Sections 3.4 and Chapter 5. 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)
as 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)
as 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 as 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) as 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)
as 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 as 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)
as 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 as 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)
as 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)
as 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) as 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) as 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 as 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 as 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 $et)
as rs:vtExtent
{
  <timestamp vtBegin="{$bt}" vtEnd="{$et}"/>
}
```

- Function `tau:periods-of()`

This function as 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) as 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 as the sequenced query results of a path expression. It is used in idiomatic slicing.

```
define function seq-path(rs:vtExtent $p, node* $TXQ, node $e)
as xs:node*
{
  typeswitch ($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 as 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)
as 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 as true when the input node is a special node (e.g., `timestamp` and `timeVaryingAttribute`) 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 as 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) as 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 as 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) as 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 as 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) as 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 ())
}
```

Appendix 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>
```

Appendix 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>
```

Appendix 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>
```

Appendix 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

```
<?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

```
<?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>
```

Appendix H
Example Instances

In this chapter, 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>
```

Appendix 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>
```

Appendix J
Mapping Result of In-Place Slicing

```
{-- validtime avg(for $c in --}
tau:apply-timestamp(
 let $tau:par1 :=
  (let $tau:s :=
    {-- let $tau:sequence:=document("CRM.xml") return --}
    (let $tau:s := (document("CRM.xml"), tau:period("1000-01-01", "9999-12-31"))
     for $tau:p in tau:const-periods2(tau:period("1000-01-01", "9999-12-31"),
                                      $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 $tau:i1 in (1 to count($tau:s1) div 2)
      let $tau:vi1 := 2 * $tau:i1 - 1
      let $tau:v1 := item-at($tau:s1, $tau:vi1)
      let $tau:p1 := item-at($tau:s1, $tau:vi1+1)
      let $tau:dot := ($tau:v1, $tau:p1) return
       {-- for $tau:dot in $tau:dot/descendant-or-self::customer return --}
       let $tau:s2 :=
        (let $tau:s3 := tau:get-actual-items($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 $tau:s2/descendant-or-self::customer
          where tau:overlaps($tau:p3,$tau:step)
          return ($tau:step, tau:intersection($tau:p3, $tau:step)))
       for $tau:i2 in (1 to count($tau:s2) div 2)
       let $tau:vi2 := 2 * $tau:i2 - 1
       let $tau:v2 := item-at($tau:s2, $tau:vi2)
       let $tau:p2 := item-at($tau:s2, $tau:vi2+1)
       let $tau:dot := ($tau:v2, $tau:p2) return
        for $tau:c in
         (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) return
            (for $tau:a in $tau:s/attribute::supportLevel return
              ($tau:a, $tau:p4),
             for $tau:ta in $tau:s/timeVaryingAttribute[@name="supportLevel]
             where tau:overlaps($tau:ta, $tau:p4) return
              ($tau:ta, tau:intersection($tau:ta, $tau:p4))))
        let $tau:s := (item-at($tau:c, 1) = "gold" or
                       item-at($tau:c, 1)/@value = "gold", item-at($tau:c, 2))
```

```
        for $tau:p in tau:const-periods2($tau:p2, $tau:s) return
         if (tau:get-actual-items(tau:sequence-in-period($tau:s, $tau:p)))
         then tau:sequence-in-period($tau:dot, $tau:p)
         else ())
   for $tau:i in (1 to count($tau:s) div 2))
   let $tau:vi := 2 * $tau:i - 1
   let $tau:v := item-at($tau:s, $tau:vi)
   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 $tau:i1 in (1 to count($tau:s1) div 2)
       let $tau:vi1 := 2 * $tau:i1 - 1
       let $tau:v1 := item-at($tau:s1, $tau:vi1)
       let $tau:p1 := item-at($tau:s1, $tau:vi1+1)
       let $tau:dot := ($tau:v1, $tau:p1) 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:p1) return
         let $tau:p4 := tau:intersection($tau:p3, $tau:p1)
         for $tau:step in $tau:s3/child::supportIncident
         where tau:overlaps($tau:p4,$tau:step)
         return ($tau:step, tau:intersection($tau:p4, $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)
```

Appendix K
XBench DC/SD Workload

- Q1 *Return the item that has matching item id attribute value (I1).*

```
document("catalog.xml")/catalog/:item[@id="I1"]
```

- Q2 *Find the title of the item which has matching author first name (Ben).*

```
for $item in document("catalog.xml")/catalog/:item
where $item/authors/autor/name/first_name = "Ben"
return $item/title
```

- Q3 *Group items released in a certain year (1990), by publisher name and calculate the total number of items for each group.*

```
for $a in distinct-values(document("catalog.xml")/catalog/:item
    [date_of_release >= "1990-01-01"]
    [date_of_release < "1991-01-01"]/publisher/name)
let $b := document("catalog.xml")/catalog/:item/publisher[name=$a]
return <Output>
          <Publisher>{$a/text()}</Publisher>
          <NumberOfItems>{count($b)}</NumberOfItems>
       </Output>
```

- Q4 *List the item id of the previous item of a matching item with id attribute value (I2).*

```
let $item := document("catalog.xml")/catalog/:item[@id="I2"]
for $prevItem in document("catalog.xml")/catalog/:item
    [. << $item][position() = last()]
return <Output>
          <CurrentItem>{$item/@id}</CurrentItem>
          <PreviousItem>{$prevItem/@id}</PreviousItem>
       </Output>
```

- Q5 *Return the information about the first author of item with a matching id attribute value (I3).*

```
for $a in document("catalog.xml")/catalog/:item[@id="I3"]
return $a/authors/author[1]
```

- Q6 *Return item information where some authors are from certain country (Canada).*

```
for $item in document("catalog.xml")/catalog/:item
where some $auth in
    $item/authors/author/contact_information/mailing_address
    satisfies $auth/name_of_country = "Canada"
return $item
```

- Q7 *Return item information where all its authors are from certain country (Canada).*

```
for $item in document("catalog.xml")/catalog/:item
where every $add in
    $item/authors/author/contact_information/mailing_address
    satisfies $add/name_of_country = "Canada"
return $item
```

- Q8 *Return the publisher of an item with id attribute value I4.*

```
for $a in document("catalog.xml")/catalog/*[@id="I4"]
return $a/publisher
```

- Q9 *Return the ISBN of an item with id attribute value I5.*

```
for $a in document("catalog.xml")/catalog/:item
where $a/@id="I5"
return $a//ISBN/text()
```

- Q10 *List the item titles ordered alphabetically by publisher name, with release date within a certain time period (from 1990-01-01 to 1995-01-01).*

```
for $a in document("catalog.xml")/catalog/:item
where $a/date_of_release gt "1990-01-01" and
      $a/date_of_release lt "1995-01-01"
order by $a/publisher/name
return <Output>
          {$a/title}
          {$a/publisher}
       </Output>
```

- Q11 *List the item titles in descending order by date of release with release date within a certain time period (from 1990-01-01 to 1995-01-01).*

```
for $a in document("catalog.xml")/catalog/:item
where $a/date_of_release gt "1990-01-01" and
      $a/date_of_release lt "1995-01-01"
order by $a/date_of_release descending
```

```
return <Output>
          {$a/title}
          {$a/date_of_release}
       </Output>
```

- Q12 *Get the mailing address of the first author of certain item with id attribute value (I6).*

```
for $a in document("catalog.xml")/catalog/:item[@id="I6"]
return
  <Output>
     {$a/authors/author[1]/contact_information/mailing_address}
  </Output>
```

- Q14 *Return the names of publishers who publish books between a period of time (from 1990-01-01 to 1991-01-01) but do not have FAX number.*

```
for $a in document("catalog.xml")/catalog/:item
where $a/date_of_release gt "1990-01-01" and
      $a/date_of_release lt "1991-01-01" and
      empty($a/publisher/contact_information/FAX_number)
return <Output>
          {$a/publisher/name}
       </Output>
```

- Q17 *Return the ids of items whose descriptions contain a certain word ("hockey").*

```
for $a in document("catalog.xml")/catalog/:item
where contains ($a/description, "hockey")
return <Output>{$a/@id}</Output>
```

- Q19 *Return the item titles related by certain item with id attribute value (I7).*

```
for $item in document("catalog.xml")/catalog/:item[@id="I7"]
    $related in document("catalog.xml")/catalog/:item
where $item/related_items/related_item/item_id = $related/@id
return <Output>
          {$related/title}
       </Output>
```

- Q20 *Retrieve the item title whose size (length * width * height) is bigger than certain number (500000).*

```
for $size in document("catalog.xml")/catalog/:item/attributes/
             size_of_book
where $size/length * $size/width * $size/height > 500000
return <Output>
          {$size/../../title}
       </Output>
```

Appendix L
Temporal Relational Schema

- `book`

Columns	Type
id	CHARACTER(10)
ISBN	CHARACTER VARYING(20)
title	CHARACTER(20)
subject	CHARACTER VARYING(200)
number_of_pages	INTEGER
type_of_book	CHARACTER(10)
length	FLOAT
length_unit	CHARACTER(10)
width	FLOAT
width_unit	CHARACTER(10)
height	FLOATl
height_unit	CHARACTER(10)
suggested_retail_price	DECIMAL(10,2)
SRP_currency	CHARACTER(10)
cost	DECIMAL(10,2)
cost_currency	CHARACTER(10)
when_is_available	DATE
quantity_in_stock	INTEGER
date_of_release	DATE
description	CHARACTER(500)
publisher_id	CHARACTER(10)
begin_time	DATE
end_time	DATE

Table L.1. The schema of book

- `author`

Columns	Type
author_id	CHARACTER(10)
first_name	CHARACTER(20)
middle_name	CHARACTER(20)
last_name	CHARACTER(20)
date_of_birth	DATE
biography	CHARACTER(500)
street_address1	CHARACTER VARYING(30)
street_address2	CHARACTER VARYING(30)
name_of_city	CHARACTER VARYING(20)
name_of_state	CHARACTER VARYING(20)
zip_code	CHARACTER(8)
name_of_country	CHARACTER VARYING(20)
phone_number	CHARACTER VARYING(20)
email_address	CHARACTER VARYING(30)
begin_time	DATE
end_time	DATE

TABLE L.2. The schema of author

- `book_author`

Columns	Type
book_id	CHARACTER(10)
author_id	CHARACTER(10)
begin_time	DATE
end_time	DATE

TABLE L.3. The schema of book_author

- `publisher`

Columns	Type
publisher_id	CHARACTER(10)
name	CHARACTER VARYING(40)
street_address1	CHARACTER VARYING(30)
street_address2	CHARACTER VARYING(30)
name_of_city	CHARACTER VARYING(20)
name_of_state	CHARACTER VARYING(20)
zip_code	CHARACTER(8)
exchange_rate	FLOAT
currency	CHARACTER(10)
phone_number	CHARACTER VARYING(20)
web_site	CHARACTER VARYING(40)
FAX_number	CHARACTER VARYING(20)
begin_time	DATE
end_time	DATE

TABLE L.4. The schema of publisher

- `related_book`

Columns	Type
book_id	CHARACTER(10)
related_id	CHARACTER(10)
begin_time	DATE
end_time	DATE

TABLE L.5. The schema of related_book

Appendix M
Non-Temporal SQL Queries with PSMs

- Q1 *Find the title of the item which has matching author first name (Ben).*

```
SELECT b.title
FROM book b, book_author ba
WHERE b.id = ba.book_id
AND get_author_name(ba.author_id) = 'Ben'

CREATE FUNCTION get_author_name(aid CHARACTER(10))
    RETURNS CHARACTER(20)
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE fname CHARACTER(20);
    set fname = (SELECT first_name
                 FROM author
                 WHERE author_id = aid;
    return fname;
  END;
```

- Q2 *Group items released in a certain year (1990), by publisher name and calculate the total number of items for each group.*

```
SELECT p.name, SUM(b.quantity_in_stock)
FROM book b, publisher p
WHERE b.publisher_id = p.publisher_id
AND release_date_between(b.id, '1990-01-01', '1991-01-01')
GROUP BY p.name;

CREATE FUNCTION release_date_between(bid CHARACTER(10), start DATE, end DATE)
    RETURNS BOOLEAN
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE rd DATE;
    CALL get_release_date(bid, rd);
    IF (rd >= start) AND (rd < end)
    THEN RETURN true;
    ELSE RETURN false;
  END;
```

```
CREATE PROCEDURE release_date(IN bid CHARACTER(10), OUT rd DATE)
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    SET rd =  (SELECT date_of_release
               FROM book
               WHERE id = bid);
  END;
```

- Q3 *Group items released in a certain year (1990), by publisher name and calculate the total number of items for each group.*

```
SELECT p.name, SUM(b.quantity_in_stock)
FROM book b, publisher p
WHERE b.publisher_id = p.publisher_id
AND release_date(b.id) >= '1990-01-01'
AND release_date(b.id) < '1991-01-01'
GROUP BY p.name

CREATE FUNCTION release_date(bid CHARACTER(10))
    RETURNS DATE
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE rd DATE;
    SET rd =  (SELECT date_of_release
                FROM book
                WHERE id = bid);
    RETURN rd;
  END;
```

- Q4 *List the item titles ordered alphabetically by publisher name, with release date in 1990s.*

```
SELECT b.title, p.name
FROM publisher p, book b
WHERE b.publisher_id = p.publisher_id
AND book_of_90s(b.id)
ORDER BY p.name;

CREATE FUNCTION book_of_90s(bid CHARACTER(10))
    RETURNS BOOLEAN
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
```

```
    DECLARE rd DATE;

    SET rd = (SELECT date_of_release
              FROM book
              WHERE id = bid);

    IF (rd >= '1990-01-01') AND (rd < '2000-01-01')
    THEN RETURN TRUE;
    ELSE RETURN FALSE;
  END;
```

- Q5 *List the item titles in descending order by date of release with more than one author and more than two related books.*

```
SELECT title, date_of_release
FROM book
WHERE multi_author_multi_realted(id)
ORDER BY date_of_release DESCENDING;

CREATE FUNCTION multi_author_multi_realted(bid CHARACTER(10))
    RETURNS BOOLEAN
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE num_a, num_r INTEGER

    set num_a = (SELECT COUNT(*)
                 FROM book_auhor
                 WHERE book_id = bid);
    set num_r = (SELECT COUNT(*)
                 FROM related_book
                 WHERE book_id = bid);

    IF (num_a > 1) AND (num_r > 2)
    THEN RETURN TRUE;
    ELSE RETURN FALSE;
  END;
```

- Q6 *Return the names of publishers that do not have FAX number and some books published by it are out of stock.*

```
SELECT p.name
FROM publisher p, book b
WHERE p.publisher_id = b.publisher_id
AND quantity(b.id) = 0
AND p.fax_number IS NULL;
```

```
CREATE FUNCTION quantity(bid CHARACTER(10))
    RETURNS INTEGER
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE qty INTEGER;
    SET qty =  (SELECT quantity_in_stock
                FROM book
                WHERE id = bid);
    RETURN qty;
  END;
```

- Q7 *Return item information where some authors are from certain country (Canada).*

```
SELECT b.*
FROM book b
WHERE some_author_from_country(b.id, 'Canada');

CREATE FUNCTION some_author_from_country(bid CHARACTER(10),
                                         country CHARACTER(20))
    RETURNS BOOLEAN
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE country_name CHARACTER(20);

    DECLARE country_cursor CURSOR FOR
    SELECT a.name_of_country
    FROM author a, book_author ba
    WHERE a.author_id = ba.author_id
    AND ba.book_id = bid;

    DECLARE country_not_found CONDITION FOR SQLSTATE '02000';

    OPEN country_cursor;
    FETCH country_cursor INTO country_name;
    WHILE NOT(country_not_found) DO
      IF (country_name = country)
      THEN
        BEGIN
          close country_cursor;
          RETURN true;
        END
      ELSE FETCH country_cursor INTO country_name;
    END WHILE
    CLOSE country_cursor;
    RETURN false;
  END;
```

- Q8 *Return item information where all its authors are from certain country (Canada).*

```
SELECT b.*
FROM book b
WHERE all_author_from_country(b.id, 'Canada');

CREATE FUNCTION all_author_from_country(bid CHARACTER(10),
                                        country CHARACTER(20))
    RETURNS BOOLEAN
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE country_name CHARACTER(20);

    DECLARE country_cursor CURSOR FOR
    SELECT a.name_of_country
    FROM author a, book_author ba
    WHERE a.author_id = ba.author_id
    AND ba.book_id = bid;

    DECLARE country_not_found CONDITION FOR SQLSTATE '02000';

    OPEN country_cursor;
    FETCH country_cursor INTO country_name;
    WHILE NOT(country_not_found) DO
      IF (country_name <> country)
      THEN
        BEGIN
          close country_cursor;
          RETURN false;
        END
      ELSE FETCH country_cursor INTO country_name;
    END WHILE
    CLOSE country_cursor;
    RETURN true;
  END;
```

- Q9 *Return the item titles related by certain item with id attribute value (I7).*

```
SELECT b.title
FROM book b, related_book rb
WHERE related_by(b.id, 'I7');

CREATE FUNCTION related_by(rid CHARACTER(10), bid CHARACTER(10))
    RETURNS BOOLEAN
    LANGUAGE SQL
    READ SQL DATA
```

```
BEGIN
  DECLARE related_id CHARACTER(10);

  DECLARE related_cursor CURSOR FOR
  SELECT related_id
  FROM related_book
  WHERE book_id = 'I7';

  DECLARE item_not_found CONDITION FOR SQLSTATE '02000';

  OPEN related_cursor;
  FETCH related_cursor INTO related_id;
  WHILE NOT(item_not_found) DO
    IF (related_id = rid)
    THEN
      BEGIN
        close related_cursor;
        RETURN true;
      END
    ELSE FETCH related_cursor INTO related_id;
  END WHILE
  CLOSE related_cursor;
  RETURN false;
END;
```

- Q10 *Retrieve the item title whose size (length * width * height) is bigger than certain number (500000).*

```
SELECT b.title
FROM book b
WHERE big_book(b.id);

CREATE FUNCTION big_book(bid CHARACTER(10))
    RETURNS BOOLEAN
    LANGUAGE SQL
    READ SQL DATA
  BEGIN
    DECLARE size NUMBER;
    SET size = (SELECT length * width * height FROM book
                WHERE id = bid);
    IF (size > 500000)
    THEN RETURN true;
    ELSE RETURN false;
  END;
```

References

[1] H. Ahlert. Enterprise Customer Mangement: Integrating Corporate and Customer Information. In *Relationship Marketing*, Springer, 2000.

[2] T. Amagasa, M. Yoshikawa, and S. Uemura. A Data Model for Temporal XML documents. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 334–344, London, England, 2000.

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

[4] J. Bair, M. Böhlen, C. S. Jensen, and R. T. Snodgrass. Notions of Upward Compatibility of Temporal Query Languages. *Business Informatics (Wirtschafts Informatik)*, 39(1):25–34, February 1997.

[5] S. Boag, D. Chamberlin, M. Fernández, D. Fernández, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, August 2003.

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

[7] T. Bohme and E. Rahm. XMach-1: A Benchmark for XML Data Management. In *Proceedings of German Database Conference (BTW)*, pages 264–273, 2001.

[8] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 second edition W3C recommendation. Technical Report REC-xml-20001006, World Wide Web Consortium, October 2000.

[9] S. Bressan, M. Lee, Y. G. Li, Z. Lacroix, and U. Nambiar. The XOO7 XML Management System Benchmark. Technical Report TR21/00, National University of Singapore, CS Department, 2001.

[10] P. Buneman, S. Khanna, K. Tajima, and W-C. Tan. Archiving Scientific Data. In *Proceedings of the 2002 ACM-SIGMOD Conference*, pages 1–12, Madison, Wisconsin, 2002.

[11] D. Chamberlin, P. Fankauser, M. Marchiori, and J. Robie. XML Query Use Cases. http://www. w3.org/TR/xmlquery-use-cases. World Wide Web Consortium.

[12] S. S. Chawathe, S. Abiteboul, and J. Widom. Managing Historical Semistructured Data. *Theory and Practice of Object Systems*, 5(3):143–162, 1999.

[13] S-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Efficient Complex Query Support for Multiversion XML Documents. In *Proceedings of International Conference on Extending Database Technology*, pages 25–27, Prague, Czech, March 2002.

[14] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proceedings of International Conference on Data Engineering*, pages 41–52, San Jose, CA, February 2002.

[15] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proceedings of the 18th Inter. Conference on Data Engineering*, pages 41–52, San Jose, CA, 2002.

[16] Microsoft Corporation. XML Query Language Demo. `http://131.107.228.20/xquerydemo`.

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

[18] F. Currim, S. Currim, C. E. Dyreson, and R. T. Snodgrass. A Tale of Two Schemas: Creating A Temporal XML Schema from a Snapshot Schema with τXSchema. In *Proceedings of International Conference on Extending Database Technology*, Heraklion-Crete, Greece, 2004.

[19] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. `http://www.w3.org/TR/2002/WD-query-semantics-20021115/`, November 2002.

[20] C. Dyreson. Observing Transaction-Time Semantics with *TT*XPath. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering*, pages 193–202, Kyoto, Japan, 2001.

[21] C. E. Dyreson, M. H. Böhlen, and C. S. Jensen. Capturing and Querying Multiple Aspects of Semistructured Data. In *Proceedings of the 25th VLDB Conference*, pages 290–301, Edinburgh, Scotland, 1999.

[22] Howard Katz (eds.), D. Chamberlin, D. Draper, M. Fernández, M. Kay, J. Robie, M. Rys, J. Siméon, J. Tivy, and P. Wadler. XQuery from the Experts: A Guide to the W3C XML Query Language. Addison-Wesley, 2003.

[23] D. C. Fallside. XML Schema Part 0: Primer . http://www.w3.org/TR/-xmlschema-0, May 2001.

[24] S. Gallant, G. Piatetsky-Shapiro, and M. Tan. Value-Based Data Mining for CRM. In *Proceedings of the 10th KDD International Conference*, San Francisco, CA, 2001.

[25] D. Gao and R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In *Proceedings of International Conference on Very Large Data Bases*, pages 632–643, Berlin, Germany, September 2003.

[26] 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*, pages 294–303, Izmir, Turkey, 2000.

[27] F. Grandi, F. Mandreoli, P. Tiberio, and M. Bergonzini. A Temporal Data Model and Management System for Normative Texts in XML Format. In *Proceedings of 5th ACM CIKM International Workshop Web Information and Data Management*, pages 29–36, New Orlean, Louisana, 2003.

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

[29] C. S. Jensen and C. E. Dyreson (eds). A Consensus Glossary of Temporal Database Concepts—February 1998 Version. Springer-Verlag, 1998.

[30] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying Temporal Models via a Conceptual Model. *Information Systems*, 19(7):513–547, 1994.

[31] Lucent Bell Lab and AT&T Research. Galax Version 0.3.1. `http://db.bell-labs.com/galax`, June 2003.

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

[33] M. H. Böhlen and R. T. Snodgrass and M. D. Soo. Coalescing in Temporal Databases. In *Proceedings of International Conference on Very Large Databases*, pages 180–191, Bombay, India, 1996.

[34] M. Manukyan and L. Kalinichenko. Temporal XML. In *Proceedings of 5th East European Conference on Advances in Databases and Information Systems*, pages 143–155, Vilnius, Lithuania, 2001.

[35] 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 Data Bases*, pages 581–590, Rome, Italy, September 2001.

[36] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-Centric Management of Versions in an XML Warehouse. In *Proceedings of the 27th VLDB Conference*, pages 581–590, Rome, Italy, 2001.

[37] Jim Melton. Understanding SQL's Stored Procedures, A complete Guide to SQL/PSM. Morgan Kaufmann, 1998.

[38] University of Washington. XML Data Repository. http://www.cs.washington.edu/research/xmldatasets/, 2002.

[39] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, August 1995.

[40] K. Runapongsa, J. M. Patel, H. V. Jagadish, and S. Al-Khalifa. The Michigan Benchmark. Technical Report, University of Michigan, 2002.

[41] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management . In *Proceedings of the 28th VLDB Conference*, pages 974–985, Hong Kong, China, 2002.

[42] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. A Foundation for Conventional and Temporal Qeury Optimization Addressing Duplicates and Ordering. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):21–49, 2001.

[43] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Adaptive Query Optimization and Evaluation in Temporal Middleware. In *Proceedings of the 2001 ACM-SIGMOD Conference*, pages 127–138, Santa Barbara, CA, 2001.

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

[45] R. T. Snodgrass. Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann Publishers, 2002.

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

[47] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and Andreas Steiner. Adding Valid Time to SQL/Temporal. Change Proposal for ISO, 1997.

[48] 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.

[49] ISO/IEC International Standard. Information Technology – Database Languages – SQL –. ISO/IEC, 1999.

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

[51] A. Tansel, J. Clifford, S. Jajodia, A. Segev, and R. T. Snodgrass (eds.). Temporal Databases: Theory, Design, and Implementation. Database Systems and Applications Series. Benjamin/Cummings, Redwood City, CA, 1993.

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

[53] F. Vitali and D. Durand. Using Versioning to support Collaboration on the WWW. *World Wide Web Journal*, 1(1):37–50, 1996.

[54] F. Wang and C. Zaniolo. Preserving and Querying Histories of XML-Published Relational Databases. In *Proceedings of the 2nd International Workshop on Evolution and Change in Data Management*, pages 26–38, Tampere, Finland, 2002.

[55] F. Wang and C. Zaniolo. Publishing and Querying the Histories of Archived Relational Databases in XML. In *Proceedings of the 4th International Conference on Web Information Systems Engineering*, pages 93–102, Rome, Italy, 2003.

[56] F. Wang and C. Zaniolo. Temporal Queries in XML Document Archives and Web Warehouses. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning*, pages 47–55, Queensland, Australia, 2003.

[57] B. B. Yao, M. T. Ozsu, and J. Keenleyside. XBench - A Family of Benchmarks for XML DBMSs. Technical Report CS-TR-2002-39, School of Computer Science, University of Waterloo, 2002.

[58] S. Zhang and C. Dyreson. Adding Valid Time to XPath. In *Proceedings of the International Workshop on Database and Network Information Systems*, pages 29–42, Aizu, Japan, 2002.