τBench: Extending XBench with Time

Stephen W. Thomas, Richard T. Snodgrass, and Rui Zhang

December 6, 2010

TR-92

A TIMECENTER Technical Report
τBench: Extending XBench with Time

All rights reserved.

Author(s) Stephen W. Thomas, Richard T. Snodgrass, and Rui Zhang

Publication History December 2010, a TIMECENTER Technical Report

TIMECENTER Participants

Michael H. Böhlen, University of Zurich, Switzerland; Curtis E. Dyreson, Utah State University, USA; Fabio Grandi, University of Bologna, Italy; Christian S. Jensen (codirector), Aarhus University, Denmark; Vijay Khatri, Indiana University, USA; Gerhard Knolmayer, University of Berne, Switzerland; Carme Martín, Technical University of Catalonia, Spain; Thomas Myrach, University of Berne, Switzerland; Mario A. Nascimento, University of Alberta, Canada; Sudha Ram, University of Arizona, USA; John F. Roddick, Flinders University, Australia; Keun H. Ryu, Chungbuk National University, Korea; Simonas Šaltenis, Aalborg University, Denmark; Dennis Shasha, New York University, USA; Richard T. Snodgrass (codirector), University of Arizona, USA; Paolo Terenziani, University of Piemonte Orientale “Amedeo Avogadro,” Alessandria, Italy; Stephen W. Thomas, Queen’s University, Canada; Kristian Torp, Aalborg University, Denmark; Vassilis Tsotras, University of California, Riverside, USA; Fusheng Wang, Emory University, USA; Jef Wijsen, University of Mons-Hainaut, Belgium; and Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.aau.dk/TimeCenter>

Any software made available via TIMECENTER is provided “as is” and without any express or implied warranties, including, without limitation, the implied warranty of merchantability and fitness for a particular purpose.

The TIMECENTER icon on the cover combines two “arrows.” These “arrows” are letters in the so-called Rune alphabet used one millennium ago by the Vikings, as well as by their predecessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote “T” and “C,” respectively.
# Contents

**Table of Contents**

1 Introduction 1

1.1 Anatomy of a Benchmark 1

1.2 Philosophy of the τBench Suite 1

2 Background 3

2.1 XBench 3

2.1.1 Data Generation 3

2.1.2 Data Size 4

2.1.3 DC/SD: The Book Store Catalog (catalog.xml) 4

2.2 τXSchema 4

2.3 τXQuery 4

2.4 Persistent Stored Modules 5

2.5 τPSM 5

3 Temporal Data Model 7

4 Architecture of the τBench Suite 9

4.1 The τX Bench Benchmark 9

4.1.1 The τX Bench-Galax Benchmark 9

4.2 The τXSchema Benchmark 10

4.3 The PSM Benchmark 10

4.3.1 The PSM-DB2 Benchmark 10

4.4 The τPSM Benchmark 10

4.4.1 The τPSM-DB2 Benchmark 10

5 Schemas 11

5.1 XBench XML Schema 11

5.2 τXBench XML Schemas and Temporal XML Schema 12

5.2.1 τXBench-Galax XML Schemas and Temporal XML Schema 12

5.3 τXSchema XML Schemas and Temporal XML Schema 12

5.4 PSM Relational Schemas 14

5.4.1 PSM-DB2 Relational Schemas 15

5.5 τPSM Relational Schemas 15

5.5.1 τPSM-DB2 Relational Schemas 15

6 Data 17

6.1 Datasets 17

6.2 XBench Non-temporal XML Data 18

6.3 τXBench Temporal XML Data 19

6.4 τXSchema Temporal XML Data 19

6.5 PSM Non-temporal Relations 19

6.6 τPSM Temporal Relations 19

6.7 Validation of Generated Data 19
# Workloads

## 7.1 XBench XQuery Queries

## 7.2 τXBench τXQuery Queries

## 7.3 τXSchema τXQuery Queries

## 7.4 PSM Queries

## 7.5 τPSM Queries

# Supporting Tool Suite

## 8.1 τGenerator: Simulating Temporal Data

## 8.2 τCorruptor: Corrupting Temporal Data

## 8.3 Validating the Correctness of Schemas, Data and Workloads

# Conclusions and Future Research

# XML Schemas

## A.1 Conventional XML Schema (DCSD.xsd)

## A.2 Temporal XML Schema

## A.3 Conventional XML Schema with Added Constraints

## A.4 Representational XML Schema

## A.5 Temporal XML Schema with Added Constraints

# Relational Schemas

## B.1 Non-temporal Relational Schemas

## B.2 Temporal Relational Schemas

# Example Parameters File for τGenerator

# DB2 differences
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The three components of a benchmark.</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>A subset of the Book Store schema.</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>A simplified slice of the temporal <code>catalog.xml</code> document.</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Overview of the τBench suite of benchmarks.</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>Overview of the conceptual relationships between the τBench schemas.</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>The cardinality constraint (C5) of XBench’s XML Schema.</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Overview of the mechanical generation of the τBench schemas.</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>The non-sequenced cardinality constraint (C5) of τXSchema’s XML Schema.</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>Cardinality constraint C5 of the PSM benchmark schema.</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>Overview of the conceptual relationships between the τBench datasets.</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>Overview of the mechanical generation of the τBench datasets.</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>Overview of the conceptual relationships between the τBench workloads.</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>Overview of mechanical generation of the τBench workloads.</td>
<td>22</td>
</tr>
<tr>
<td>14</td>
<td>Query Q2 of the XBench benchmark.</td>
<td>22</td>
</tr>
<tr>
<td>15</td>
<td>Query Q2 of the τXQuery benchmark, with a <code>validtime</code> statement modifier.</td>
<td>22</td>
</tr>
<tr>
<td>16</td>
<td>Query Q2 of the PSM benchmark.</td>
<td>24</td>
</tr>
<tr>
<td>17</td>
<td>Query Q2 of the PSM-DB2 benchmark.</td>
<td>25</td>
</tr>
<tr>
<td>18</td>
<td>Query Q2 of the τPSM benchmark.</td>
<td>25</td>
</tr>
<tr>
<td>19</td>
<td>Query Q2 of the τPSM-DB2 benchmark.</td>
<td>26</td>
</tr>
<tr>
<td>20</td>
<td>Overview of the τGenerator process.</td>
<td>27</td>
</tr>
<tr>
<td>21</td>
<td>Two versions of a simplified <code>&lt;author&gt;</code> temporal element.</td>
<td>31</td>
</tr>
<tr>
<td>22</td>
<td>The <code>item</code> table creation and constraint definitions in SQL.</td>
<td>57</td>
</tr>
<tr>
<td>23</td>
<td>The <code>author</code> table creation and constraint definitions in SQL.</td>
<td>57</td>
</tr>
<tr>
<td>24</td>
<td>The <code>item_author</code> table creation and constraint definitions in SQL.</td>
<td>57</td>
</tr>
<tr>
<td>25</td>
<td>The <code>item_publisher</code> table creation and constraint definitions in SQL.</td>
<td>58</td>
</tr>
<tr>
<td>26</td>
<td>The <code>publisher</code> table creation and constraint definitions in SQL.</td>
<td>58</td>
</tr>
<tr>
<td>27</td>
<td>The <code>related_items</code> table creation and constraint definitions in SQL.</td>
<td>58</td>
</tr>
<tr>
<td>28</td>
<td>The <code>item</code> table creation and constraint definitions in SQL.</td>
<td>59</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The implicit and explicit schema constraints in the XBench benchmark.</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>The additional schema constraints in the $\tau$XSchema benchmark.</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>The characteristics of DS0.&quot;. DS0.HUGE is excluded since it is too large to run in the current implementation of $\tau$Generator. Note that the sizes of the input catalog.xml files and the output output_date.xml files differ, even though they have the same content, because $\tau$Generator outputs the XML files with indentation, whereas XBench output is more compressed.</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>The characteristics of the LARGE class of the four temporal datasets we define in $\tau$Bench.</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>The features highlighted by the queries in the XBench Benchmark. Reproduced from [1].</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>The features highlighted by the queries in the PSM Benchmark.</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>The inputs into $\tau$Generator.</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>The output files of $\tau$Generator.</td>
<td>31</td>
</tr>
<tr>
<td>9</td>
<td>The two versions of the &lt;author&gt; temporal element, shredded into two tuples.</td>
<td>31</td>
</tr>
</tbody>
</table>
1 Introduction

Time is God’s way of keeping everything from happening at once.
-Anonymous

Since the advent of the computer in the 1960s, humans have recorded increasingly large amounts of data into various digital formats, such as relational databases, spreadsheets, and flat text files. The extensible markup language (XML) has become one of the more popular methods for storing and exchanging documents and data [2], due to its flexibility and self-describing nature. One source of evidence for XML’s popularity is the huge number of tools that have recently been developed to store, maintain, query, and validate XML documents.

XBench is a family of benchmarks typically used to evaluate existing XML tools [3, 4, 1]. The authors of XBench analyzed several real-world datasets to measure their characteristics (such as typical data types and length of records), and the XBench datasets reflect these measurements. Having such a benchmark creates an environment for fair, accurate, realistic, and reproducible comparisons amongst competing XML tools and algorithms.

However, XBench only provides a single snapshot of a dataset, even though the data typically modeled in XML is known to change over time: companies evolve with new products and employees; new scientific discoveries are made; stock markets fluctuate on a minute-by-minute basis; and social networks are expanded. As such, temporal (or time-varying) data naturally arises in many XML applications. A temporal XML document records the evolution of an XML document over time, i.e., all of the versions of the document. Capturing a document’s evolution is vital to supporting time travel queries that delve into a past version, and incremental queries that involve the changes between two versions.

Despite the prevalence of temporal data in real-world applications, no temporal benchmarks currently exist. Several strategies for managing temporal XML documents have been proposed [5, 6, 7, 8, 9, 10], but there is no easy way to compare them in a practical setting. DBMS vendors are starting to add (limited) temporal support, but again, there is no standard mechanism for comparing the performance of two DBMSes.

To create a standard comparison mechanism for temporal applications, we introduce τBench, a temporal extension to the non-temporal XBench benchmark. τBench consists of a suite of temporal and non-temporal benchmarks, all derived from XBench, as well as a suite of tools for generating and validating each benchmark. We describe our goals for creating such a suite and our mechanical processes for generating and validating each benchmark. We have made the τBench suite of benchmarks and tools publicly available [11].

In addition to a suite of benchmarks, τBench can be viewed as a general framework for creating benchmarks, as it provides a simple and robust mechanism for the creation and validation of new benchmarks from existing benchmark components. In this report, emphasis is placed on validation throughout the τBench process: schemas, datasets, and workloads are evaluated for correctness at several points along the creation process.

A cornerstone of the τBench tool suite is a temporal data generation tool, τGenerator, which can randomly generate time-varying documents in several data formats. We describe the user-specified inputs into the tool and describe the internal simulation that randomly changes data elements at specified time intervals.

Finally, we describe an initial set of benchmarks that we have created in τBench, which currently subsumes technologies such as XML Schema, XQuery, τXQuery, τXSchema, persistent stored modules (PSM), and τPSM.

1.1 Anatomy of a Benchmark

As depicted in Figure 1, a benchmark consists of three components: data, schema for the data, and workloads to perform on the data. This general definition of a benchmark allows for the data to take on various forms and formats, the schema to loosely or tightly constrain the characteristics of the data, and any sort of workload to be performed (for example, queries, constraint checking, or bulk loads).

To define a benchmark, one must define all three components of the benchmark. Providing only one of the components without the others does not define a usable benchmark and is therefore of limited value. In this report, we introduce a suite of benchmarks, each with these three components.

1.2 Philosophy of the τBench Suite

In creating the τBench suite of benchmarks, we had several goals in mind. First and foremost, we wanted to define a well-structured framework of benchmarks, such that the various components of each benchmark could be
shared with or derived from other benchmarks. This not only allowed a modular and simple construction of each benchmark, but it also provided a mechanism for creating new benchmarks from existing components, similar to frameworks in the Java programming language.

Our second goal was to exploit the use of statement modifiers for transforming the components of one benchmark into another [12]. A statement modifier (such as SQL’s \texttt{ALTER TABLE} statement) has the advantages of simplicity of implementation and understanding, ease of validation (by visual inspection), and offloading of functionality to existing tools. For example, by simply adding an \texttt{ALTER TABLE} statement modifier, the user is offloading the modification of the schema to the DBMS, instead of modifying the schema by hand, which can be difficult and error-prone. It is easy to validate that the newly created schemas are correct, because only a few statements were added to the schema definition—no complicated manual changes are necessary. As long as the DBMS’s interpretation of the statement modifiers is correct, the resulting schemas will be correct.

Our third goal was to build a robust and correct benchmark. The correctness of a benchmark is paramount to its utility in research and industry, and thus we placed great importance on our validation processes. We have built a suite of tools and processes for validating each component of each benchmark, as we describe throughout this report. In particular, we achieve a triangulation of validation by validating the interrelated components of a benchmark by different means and from different perspectives, thereby testing the same code several ways.

For example, one of the benchmarks in $\tau$Bench provides data, schemas, and workloads that can be used to test $\tau$XQuery, a temporal extension to XQuery. During the creation of this benchmark, the generated temporal data is validated by XMLLint (for physical structure), $\tau$XSchema (for logical validity), and our own tools (to ensure the content of temporal document matches its corresponding slices); the temporal schema for the temporal data is validated by XMLLint (for XML well-formedness) and $\tau$XMLLint (to ensure the constraints are well-formed and sensible); and the XML data slices are validated against their conventional schema by XMLLint. We also have a data-corruption tool, $\tau$Corruptor, to test the results of the individual temporal queries. Hence each component is tested from various angles, and must perform flawlessly on all tests. Each added test, even those added downstream, improve the quality to the entire benchmark suite, because of the interconnectedness of all of the components.

A fourth goal was to define benchmark components that were independent of a specific DBMS, by exploiting available standards, then derive DBMS-specific components via a small number of transformations. For example, we provide a benchmark component based on the Persistent Stored Module part of the SQL standard, then derive a closely-related benchmark component for DB2, targeted to its specific syntax. Any tests of correctness of the PSM-DB2 component help test the more generic PSM benchmark.

Finally, our last goal was to start with a standard, well-known benchmark and derive from it new benchmarks, thus building upon a solid foundation. These new benchmarks can be used with high levels of confidence for evaluating newly proposed temporal extensions and vendor-specific implementations.
2 Background

In this section we describe XBench, which serves as the baseline model for the τBench suite of benchmarks. We also describe τXSchema (a temporal extension of XMLSchema), which we will use to validate temporal XML data; τXQuery (a temporal extension of XQuery), which we will use to query temporal XML data; Persistent Stored Modules (PSM), which is a server-side SQL technology for which we will create two benchmarks; and finally τPSM, a temporal extension of PSM.

2.1 XBench

XBench is a family of benchmarks for XML database management systems (DBMSes), as well as a data generation tool for creating the benchmarks [1]. The XBench benchmarks have been widely used as standard datasets to test the performance and functionalities of new and existing XML tools and DBMSes.

To create realistic and appropriate datasets, the authors of XBench analyzed several real-world XML datasets to quantify their characteristics and relationships. As a result of this analysis, XBench characterizes database applications along two dimensions: application characteristics and data characteristics. Application characteristics indicate whether the database is data-centric or text-centric. 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 such as articles or blogs. In terms of data characteristics, two classes are identified: single document (SD) and multiple document (MD). In the single document case, the database consists of a single document with complex structures, while the multiple document cases contain a set of XML documents. Thus, XBench consists of four different categories that cover DC/SD, DC/MD, TC/SD, and TC/MD respectively.

1. **TC/SD**. The text-centric, single-document category is represented by a single document that is text-dominated. This category contains repeated similar entries, deep nesting, and references between entries. This category is based on analysis of the Oxford English Dictionary [13] and the GNU version of The Collaborative International Dictionary of English (GCIDE) [14].

2. **TC/MD**. The text-centric, multiple-document category is represented by numerous small documents that are text-dominated. This category contains references between elements and recursive elements. This category is based on analysis of the Reuters news corpus and Springer digital library.

3. **DC/SD**. The document-centric, single-document category is represented by a single document that contains little text. This category includes mostly transactional data where the element tags are more descriptive and contain less text content. This category is based on the analysis of TPC-W [15] benchmark.

4. **DC/MD**. The document-centric, multiple-document category is represented by five documents that each contain little text. Like the DC/SD category, the DC/MD category includes mostly transactional data where element tags are more descriptive and contain less text content and is based on the analysis of TPC-W benchmark.

For the initial purposes of τBench, we use the DC/SD category of documents from XBench since valid-time concepts are more applicable than in the other three categories.

2.1.1 Data Generation

XBench populates each data element and attribute with random data obtained from ToXgene [16]. ToXgene is a template-based tool that generates synthetic XML documents according to one or more templates.

The actual text content of each data element or attribute is a randomly generated string or number, depending on the specification in the ToXgene template. For example, an instance of the `<last_name>` subelement of an `<author>` could be the string “BABABABAOGREAT”. Thus, in the general case, it is not useful to consider or analyze the actual data content of the elements and attributes. However, a `<related_ID>` element will always point to an ID of the correct form (i.e., the letter “I” followed by a positive integer); other schema constraints can be enforced within ToXgene.
2.1.2 Data Size

For scalability purposes, XBench has defined four data size classes for each of the categories: small (10MB), normal (100MB), large (1GB), and huge (10GB).

2.1.3 DC/SD: The Book Store Catalog (catalog.xml)

The DC/SD XML document created by XBench, called catalog.xml, contains information about a book store. The full XML Schema is listed in Appendix A.1; an important subset is shown in Figure 2. In short, <item> (i.e., book) elements contain a list of <author>s, a <publisher>, and a list of <related_items> of related books.

It is worth emphasizing that there is a strict one-to-many relationship between <item>s and <author>s: although an item can have several authors, an author is only the author of a single item (no two items share an author). Similarly, there is a strict one-to-one relationship between <item>s and <publisher>s: an item has exactly one publisher, and each publisher publishes exactly one item.

2.2 $\tau$XSchema

$\tau$XSchema (Temporal XML Schema) is a language and set of tools that enable the construction and validation of temporal XML documents [5, 17, 18, 19, 20]. $\tau$XSchema extends XML Schema with the ability to define temporal element types. A temporal element type denotes that an element can vary over time, describes how to associate temporal elements across slices (or snapshots, which are individual versions of a document), and provides some temporal constraints that broadly characterize how a temporal element can change over time.

In $\tau$XSchema, any element type can be turned into a temporal element type by including a simple logical annotation (stating whether an element or attribute varies over valid time or transaction time, whether its lifetime is described as a continuous state or a single event, whether the item itself may appear at certain times and not at others, and whether its content changes) in the type definition. So a $\tau$XSchema document is just a conventional XML Schema document with a few annotations.

$\tau$XSchema provides a suite of tools to construct and validate temporal documents. A temporal document is validated by combining a conventional validating parser with a temporal constraint checker. To validate a temporal document, a temporal schema is first converted to a representational schema, which is a conventional XML Schema document that describes how the temporal information is represented. A conventional validating parser is then used to validate the temporal document against the representational schema. Then, the temporal constraint checker is used to validate any additional temporal constraints specified by the user in the temporal schema.

2.3 $\tau$XQuery

$\tau$XQuery adds temporal support to XQuery [21] by extending its syntax and semantics [22]. $\tau$XQuery moves the complexity of handling time from the user/application code to the query processor.

Figure 2: A subset of the Book Store schema.
In particular, $\tau$XQuery adds two temporal statement modifiers to the XQuery language: current and sequenced. A current query is a query on the current state of the XML data (i.e., elements and attributes that are valid now) and has the same semantics as a regular XQuery query applied to the current state of the XML data. Sequenced queries, on the other hand, are queries applied to each point in time, resulting in a sequence of temporal elements. Finally, $\tau$XQuery also has representational (also termed non-sequenced) queries, which query the time-line of the XML data irrespective of time. No statement modifiers are needed for representational queries.

2.4 Persistent Stored Modules

Persistent stored modules (PSM) is the Turing-complete portion of SQL in which stored programs are compiled and stored in a schema and can be executed on the SQL server by queries [23, 24]. PSM consists of stored procedures and stored functions, which are collectively referred to as stored routines. The SQL/PSM standard [23] provides a set of control-flow constructs (features) for SQL routines.

2.5 $\tau$PSM

$\tau$PSM extends PSM to allow temporal queries [22]. The approach requires minor new syntax beyond that already in SQL/Temporal to define PSM routines. $\tau$PSM enables current, sequenced, and non-sequenced semantics of queries and of PSM routines to be realized.
3 Temporal Data Model

For the purpose of modeling temporal data in τBench, we adopt the temporal data model employed by τXSchema [5]. In τXSchema, the history of an XML document is composed of individual slices (or snapshots), which are members of a sequenced set of non-overlapping versions of a temporal (or time-varying) XML document. Each slice is associated with a time period during which it is valid. The time period of a slice does not overlap with the time period of any other slice.

Individual elements within the XML document that change over time are called temporal elements. Temporal elements are represented as items, which contain a series of versions with associated timestamps. A new version of an item is created whenever any part of the item changes, including its children (unless the changed child is itself a temporal element). Thus, to extract a slice of a temporal XML document at time \( t \), we just select the version associated with time \( t \) of all temporal elements.

In this report, we chose to create four temporal elements out of the following four original catalog.xml elements: <item>, <author>, <publisher>, and <related_items>. These elements give us flexibility for defining the workloads of our benchmarks in later sections. An outline of the resulting temporal representation of XBench’s catalog.xml document is shown in Figure 3.

```xml
<catalog>
  <item_RepItem>
    <item_Version begin="2006-01-01" end="2009-07-05">
      <item>
        ...
        <author_RepItem>
          <author_Version begin="2006-01-01" end="2007-07-05">
            <author>
              ...
            </author>
          </author_Version>
          ...
        </author_RepItem>
        ...
      </item_Version>
    </item_RepItem>
    ...
  </item_RepItem>
  ...
</catalog>
```

Figure 3: A simplified slice of the temporal catalog.xml document.

In τXSchema, the time period of all temporal subelements must be contained in the time period of their parent temporal elements. Using the example above, the time period of an <author_Version> must be contained in its associated <item_Version>.

Adopting the temporal data model used by τXSchema not only brings the advantages of the data model itself, but it also allows us to use the τXSchema tool suite to construct and validate our benchmarks.
4 Architecture of the τBench Suite

Our overarching goal for creating the τBench suite of benchmarks was to produce an ecosystem of related benchmarks, where data, schema, and workloads flow from one benchmark to another, resulting in a strong, coherent, and coupled suite of benchmarks. Such an ecosystem brings several benefits. First, it encourages the re-use of components between benchmarks. Second, it allows researchers to work with the same benchmarks with different technologies. Third, it brings confidence that each benchmark is robustly built and validated.

In the subsections below, we present four new benchmarks in τBench, three of which are built for temporal data. For each benchmark, we briefly outline the motivation for creating the benchmark. Details about how the components of a benchmark are created are described in Sections 5–7. The actual data, schemas, queries, and supporting scripts for each benchmark are available in the τBench package, which can be obtained from the TimeCenter web page [11].

Each benchmark has a general definition and a set of platform- or tool-specific implementations. For example, we could define a set of SQL queries for a set of relations, but the data format and syntax may need to be altered to execute those queries using a specific DBMS. τBench provides both the general definitions and, in some cases, the specific implementations.

For each of the defined benchmarks below, we name the general benchmark of a given target according to the form:

- The target Benchmark (e.g., The PSM Benchmark)

and we name a specific implementation of the benchmark of a given target according to the form:

- The target-implementation Benchmark (e.g., The PSM-DB2 implementation).

We encourage researchers to create and share their own (general or implementation) benchmarks based on the τBench framework.

Figure 4 shows the high-level relationship between the benchmarks in the τBench suite. The suite begins with XBench, which defines a non-temporal XML dataset, the corresponding schema, and a set of 20 XQuery queries. We transform the data, schema, and queries into PSM format to create the PSM benchmark. We apply a temporal simulation (described in Section 8) to the XBench data to create the τXBench benchmark. We transform the data from the τXBench benchmark into the relational model to create the τPSM benchmark. Finally, we add additional temporal constraints to the τBench benchmark to create the τXSchema benchmark.

An important concept that we exploit during our validation process is the use of statement modifiers [12] in schemas and workloads. Statement modifiers allow us to make simple, understandable changes to existing schemas and workloads while achieving all of the desired functionality.

As previously mentioned, associated with some of these benchmarks are DBMS-specific implementation benchmarks. So from the τXBench benchmark is derived the τXBench-Galax benchmark, from the PSM benchmark is derived the τPSM-DB2 benchmark, and from the τPSM benchmark is derived the τPSM-DB2 benchmark.

We now describe each benchmark in more detail.

4.1 The τXBench Benchmark

We introduce the τXBench Benchmark, which is a direct temporal extension to the XBench benchmark. In particular, we use a temporal simulation to create temporal data from the non-temporal XBench data; we use τXSchem to create temporal and representational schemas for the temporal data; and we use τXQuery [25, 26] to create temporal queries from the XBench query set. Hence, this temporal benchmark serves as an extension to the original XBench benchmark.

4.1.1 The τXBench-Galax Benchmark

This benchmark is an implementation of the τXBench benchmark for the Galax [27] XQuery engine. Galax is an XQuery interpreter, and so can handle the schema and XQuery expressions of XBench, and thus can handle the schema and τXQuery expressions of τXBench, except for the temporal statement modifiers. There is work ongoing to provide source-to-source translators, mapping τXQuery expressions containing these temporal statement modifiers into equivalent XQuery expressions without such modifiers. When such a translator is available, the mapped τXQuery expressions will be added to the benchmark.
4.2 The τXSchematic Benchmark

We introduce the τXSchematic Benchmark, which is an extension to the τXBench benchmark. The τXSchematic benchmark provides additional (sequenced and non-sequenced) XML schema constraints for the temporal XML data, which is useful for evaluating the performance of XML schema validators, such as XMLLint [28] and τXMLLint [5].

4.3 The PSM Benchmark

We introduce the PSM Benchmark, which is a relational extension to the XBench benchmark. The PSM Benchmark consists of a relational translation of the data, schemas, and queries of XBench, which will serve as a non-temporal benchmark for PSM.

4.3.1 The PSM-DB2 Benchmark

This benchmark is an implementation of the PSM Benchmark for the IBM DB2 [29] DBMS. The (minor) differences between PSM and DB2 are listed in Appendix ??.

4.4 The τPSM Benchmark

We introduce the τPSM benchmark, which is both a temporal extension to the PSM benchmark and a relational extension to the τBench benchmark. The τPSM benchmark provides temporal relations and temporal PSM queries, which are useful for evaluating the recently-proposed τPSM language extension [22].

4.4.1 The τPSM-DB2 Benchmark

This benchmark is an implementation of the τPSM Benchmark for the IBM DB2 [29] DBMS. The differences between PSM and DB2 are listed in Appendix D. Note that DB2 doesn’t yet fully support temporal statement modifiers, so the resulting expressions cannot yet be validated. One possibility is a source-to-source translator that outputs conventional DB2 expressions, which could then be validated.
5 Schemas

In this section we describe the schemas of each benchmark. The schemas are closely related and all stem from the XBench XML schema, as depicted in Figure 5. (Compare this with Figure 1 on page 2, in which the benchmarks themselves are in the same place but the transitions are somewhat different.)

We begin with the unmodified XBench XML schema, DCSD.xsd. This schema provides basic constraints on the non-temporal XML data, such as how many subelements each element can have (e.g., an <author> can have 0 or 1 <fax_number>s), the data types of each element (e.g., <zip_code> is a string), and the uniqueness of some elements (e.g., each <item>'s id attribute must be unique). As depicted in Figure 7, from the original XBench we derive two additional XML schemas for the temporal benchmarks: a non-temporal relational schema that describes the physical structure of the temporal XML data (such as items and versions [5]), and a temporal schema that lists the temporal constraints.

5.1 XBench XML Schema

The original XBench XML Schema (DCSD.xsd) is given in Appendix A.1. Table 1 summarizes the (implicit and explicit) constraints defined by DCSD.xsd. (For simplicity, the table does not list data type constraints nor implied (default) cardinality constraints (i.e., all sub-elements of an element must occur exactly once.) It is these constraints that we preserve as we derive the schemas for the other benchmarks. We also observe and maintain these constraints during the temporal simulation. (Details are provided in Section 8.1.)

As a running example, consider Figure 6. The listing shows the XML Schema cardinality constraint that says: “Each <authors> element must have between one and four <author> subelements.” The constraint is specified via the minOccurs and maxOccurs attributes of the <element> element of XML Schema.

![Figure 5: Overview of the conceptual relationships between the τBench schemas.](image)

![Figure 6: The cardinality constraint (C5) of XBench’s XML Schema.](image)
5.2 τXBench XML Schemas and Temporal XML Schema

What We Create We define three schemas for the τXBench benchmark: a conventional XML schema to describe each XML slice, a temporal schema to provide temporal constraints for the temporal XML document, and a representational schema to describe the physical representation of the generated temporal document.

The conventional, temporal, and representational schemas are given in Appendices A.1, A.2, and A.4, respectively.

How We Create It The conventional schema is duplicated from the XBench benchmark, because the XML data slices are of identical form.

Under the τXSchem framework, a temporal schema contains three aspects: a time-ordered list of relevant conventional schemas; a set of logical annotations which describe sequenced and non-sequenced constraints on the data; and a set of physical annotations that describe how to represent the temporal data. We generate the temporal XML schema by setting DCSD.xsd as the sole conventional schema; not specifying any logical annotations (and therefore using the sequenced semantics of each of the original constraints in the conventional schema); and placing the physical timestamps at the <item>, <author>, <publisher>, and <related_author> elements, since that’s how our data generation tool works (Section 8.1).

The representational schema is automatically created using SCHEMAMAPPER, which is part of the τXSchem tool suite. The representational schema is derived from the conventional schema and physical annotations.

How We Validate It We need to validate each of the schemas listed above.

Since the conventional schema is duplicated from the XBench schema, it does not require validation.

The temporal schema (Appendix A.2) contains three parts: (i) a reference to the conventional schema, just discussed, (ii) a set of logical annotations, which specify that only <item>, <author>, <publisher>, and <related_item> elements can change, and (iii) physical annotations, which in this case are empty, which has the behavior of placing the transaction-time timestamp at each of the logical items present.

It is important to note that even though the logical annotations do not contain any explicit temporal constraints, there are implicit constraints present. By design, a temporal schema preserves the sequenced semantics of its conventional schemas. Thus, the semantics of the original DCSD.xsd schema, and in particular its integrity constraints, are preserved in the temporal schema. As an example, the non-temporal constraint in Figure 6 would still apply to each of the XML data slices, and thus at each point of time of the temporal document.

The representational schema describes how the temporal data is represented (e.g., <item_ReplItem> and <item_Version>) and is created automatically by the τXSchem tool suite based on the temporal XML Schema.

All three of these schemas are “validated” by triangulation of the validation of the data: if each slice of a temporal document is validated against the conventional schema, and if the temporal document is validated against the temporal schema, that helps ensure that the temporal schema and representational schemas are themselves valid. Also, the temporal schema is validated against its XML Schema schema. In testing, we frequently encountered validation failures of the data that indicated subtle problems in other components, whether the generation code or one of the schema specifications. Each of these failures gave us reassurance that the sum total of the validation required a high degree of correspondence and consistency between the many parts.

5.2.1 τXBench-Galax XML Schemas and Temporal XML Schema

There are no changes required for the Galax implementation of the τXBench schemas.

5.3 τXSchem XML Schemas and Temporal XML Schema

The schemas for the τXSchem benchmark are very similar to those the τXBench benchmark.

What We Create We define three schemas for the τXSchem benchmark: a conventional XML schema to describe each XML slice, a temporal schema to provide temporal constraints for the temporal XML document, and a representational schema to describe the physical representation of the generated temporal document.
The conventional, representational, and temporal schemas are given in Appendices A.3, A.4, and A.5 respectively.

**How We Create It**  
The conventional, representational, and temporal XML Schemas are first duplicated from the $\tau$Bench benchmark. We then manually add seven additional constraints as defined in Table 2. These constraints provide a simple but comprehensive set for testing the various types of constraints (referential integrity, data type, identity, and cardinality) with the various time modifiers (sequenced and non-sequenced).

For example, Figure 8 shows the non-sequenced constraint: "In any given year, an <item> may have up to six <authors>s". The constraint is specified using the syntax of $\tau$XSchema [5]. The constraint uses the minOccurs and maxOccurs attributes of the <nonSeqCardinality> element to give value bounds for the cardinality constraint. Additionally, the dimension attribute specifies the time dimension (valid time or transaction time), the evaluationWindow attribute gives the time window over which the constraint should be checked, and the slideSize attribute gives the size of slide between each successive evaluation window. In this case, since evaluationWindow and slideSize are both set to "year", the constraint is checked once for each calendar year. Should the slideSize have been set to, say, one day, then the constraint would be checked 364 times per calendar year: once for the fiscal year starting on January 1, once for the fiscal year starting January 2, etc.

**How We Validate It**  
As the conventional schema is duplicated, it does not require validation.

The temporal XML schema is the same as the temporal XML schema for $\tau$Bench, except that it has some additional sequenced and non-sequenced constraints. These constraints are defined in the $\tau$XSchema language [5].
Table 1: The implicit and explicit schema constraints in the XBench benchmark.

<table>
<thead>
<tr>
<th>Time</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Cardinality</td>
<td>Implicitly The &lt;related_items&gt; subelement of &lt;item&gt; must occur exactly once.</td>
</tr>
<tr>
<td>C2</td>
<td>Cardinality</td>
<td>Implicitly The &lt;authors&gt; subelement of &lt;item&gt; must occur exactly once.</td>
</tr>
<tr>
<td>C3</td>
<td>Cardinality</td>
<td>Implicitly The &lt;publisher&gt; subelement of &lt;item&gt; must occur exactly once.</td>
</tr>
<tr>
<td>C4</td>
<td>Cardinality</td>
<td>Explicitly There must be at least one &lt;item&gt; element.</td>
</tr>
<tr>
<td>C5</td>
<td>Cardinality</td>
<td>Explicitly The &lt;authors&gt; element must have between 1 and 4 &lt;author&gt; subelements.</td>
</tr>
<tr>
<td>C6</td>
<td>Identity</td>
<td>Explicitly The id attribute of &lt;author&gt; must be unique.</td>
</tr>
<tr>
<td>C7</td>
<td>Cardinality</td>
<td>Explicitly The &lt;FAX_number&gt; subelement of &lt;publisher&gt; must occur 0 or 1 times.</td>
</tr>
<tr>
<td>C8</td>
<td>Cardinality</td>
<td>Explicitly The &lt;related_item&gt; subelement of &lt;related_items&gt; must occur between 0 and 5 times.</td>
</tr>
<tr>
<td>C9</td>
<td>Cardinality</td>
<td>Explicitly The &lt;street_address&gt; subelement of &lt;street_information&gt; must occur 1 or 2 times.</td>
</tr>
</tbody>
</table>

Table 2: The additional schema constraints in the τXSchema benchmark.

<table>
<thead>
<tr>
<th>Time</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Sequenced</td>
<td>Cardinality An &lt;item&gt; must have between 1 and 4 &lt;author&gt;s.</td>
</tr>
<tr>
<td>C2</td>
<td>Sequenced</td>
<td>Referential Integrity A &lt;related_item&gt; should refer to a valid &lt;item&gt;.</td>
</tr>
<tr>
<td>C3</td>
<td>Sequenced</td>
<td>Identity The &lt;ISBN&gt; of an &lt;item&gt; are unique.</td>
</tr>
<tr>
<td>C4</td>
<td>Sequenced</td>
<td>Data type The &lt;number_of_pages&gt; of an &lt;item&gt; must be of type short.</td>
</tr>
<tr>
<td>C5</td>
<td>Non-sequenced</td>
<td>Cardinality Over a period of a year, an &lt;item&gt; may have up to 6 &lt;author&gt;s.</td>
</tr>
<tr>
<td>C6</td>
<td>Non-sequenced</td>
<td>Referential Integrity A &lt;related_item&gt; should refer to a valid &lt;item&gt; (possibly not currently in print).</td>
</tr>
<tr>
<td>C7</td>
<td>Non-sequenced</td>
<td>Identity An &lt;item&gt; id is unique and may not ever be re-used.</td>
</tr>
</tbody>
</table>

and thus are validated automatically by the τXSchema tool suite.

Similar to the τXBench benchmark, the representational schema for τXSchema is automatically generated from the τXSchema tool suite, and thus does not require manual validation.

5.4 PSM Relational Schemas

For this benchmark, we need a relational schema, rather than an XML schema.

What We Create We manually define a set of relational schemas that correspond to the data and constraints of the XBench schema. Specifically, we consider the shredding process of the <item>, <author>, <publisher>, and <related_items> elements of the XML data. (See Section 8.1.4 for details about the shredding process.) We define six relations (item, author, publisher, related_items, item_author, and item_publisher), primary and foreign keys for each table, and assertions to capture the cardinality constraints specified in the XBench schema.

Appendix B.1 gives the schemas (i.e., the table definitions, keys, and assertions) for ensuring the validity of the constraints defined in XBench’s XML schema.

How We Create It The relational schemas mimic their counterparts found in the XML Schema as closely as possible. The tables schemas were created by closely examining the original DCSD.xsd schema and noting its hierarchical structure and relationships. We then defined a set of corresponding tables that captured all of the information in the original XML schema. For example, if the <author> element in the XML data had a subelement named <name_of_city> of type DATE, then we would define a column in the author table named name_of_city and set the type to DATE.

We define a primary key in each relation as follows.

- item. The id column.
- author. The author_id column.
- publisher. The publisher_id column.
- related_items. The item_id and related_id columns.
- item_author. The item_id and author_id columns.
- item_publisher. The item_id and publisher_id columns.
We define foreign keys in each relation as follows.

- **related_items.** The `item_id` and `related_id` columns both refer to an existing `id` in the `item` table.
- **item_author.** The `item_id` refers to an existing `id` in the `item` table and the `author_id` refers to an existing `author_id` in the `author` table.
- **item_publisher.** The `item_id` refers to an existing `id` in the `item` table and the `publisher_id` refers to an existing `publisher_id` in the `publisher` table.

An SQL assertion was created for each of the original cardinality constraints [30]. For example, Figure 9 shows the cardinality constraint C5 of XBench as an SQL assertion.

**How We Validate It**  
To validate the table schemas, we load the data and tables into the DBMS. If the data loads successfully (during which assertions such as in Figure 9 are checked by the DBMS), then we know the schemas are valid.

For further validation, we have written a tool called τCorruptor (Section 8.2) to randomly corrupt the data in the relations in such a way that one of the pre-defined constraints is violated. We run τCorruptor to corrupt the data and load it into the DBMS. If the schemas catch the violation, then we can be sure that the constraints are working correctly.

**5.4.1 PSM-DB2 Relational Schemas**

We have modified the PSM schemas to satisfy DB2 syntax.

**5.5 τPSM Relational Schemas**

**What We Create**  
We define a set of relational schemas that correspond to the data and constraints of the XBench schema. The schemas for the τPSM benchmark are given in Appendix B.2.

**How We Create It**  
The τPSM schemas are derived directly from the PSM schemas. We extend each table with a simple temporal statement modifier (i.e., `ALTER TABLE ADD VALIDTIME ...`) to make the table time-varying. In addition, we extend each primary key to include the `begin_time` column and extend each assertion to be a sequenced assertion. For example, Figure 10 shows the modification of the SQL assertion corresponding to constraint C5.

**How We Validate It**  
As we only added simple statement modifiers to the schemas, no validation is required.

**5.5.1 τPSM-DB2 Relational Schemas**

We have modified the τPSM schemas to satisfy DB2 syntax.
6 Data

In this section we describe our process for generating the data for each benchmark. Figure 11 shows the relationships between the datasets. (Compare this with Figure 1 on page 2, in which the benchmarks themselves are in the same place but the transitions are somewhat different.) Figure 12 shows how the datasets are generated.

To aid us in the data generation process, we have developed a temporal simulation, called $\tau$Generator, which we describe in detail in Section 8. For now, it is sufficient to know that $\tau$Generator takes as input the non-temporal XML data from XBench, along with some user-defined simulation parameters, and outputs four sets of files: a temporal XML document; a set of non-temporal XML slices that correspond to the temporal XML document; a set of shredded temporal relations; and a set of shredded non-temporal relation slices that correspond to the temporal relations. Various combinations of these files make up the datasets for each of our benchmark, as we describe below.

![Figure 11: Overview of the conceptual relationships between the $\tau$Bench datasets.]

6.1 Datasets

We define four datasets, each of which is generated by $\tau$Generator, three of which are temporal, and all of which are summarized in Tables 3 and 4. Each dataset can come in any size, depending on the class of the input XBench dataset (small, medium, large, or huge). We name each dataset DS#.class, where '#' is a unique identifier for the dataset and class is the class of the input XBench dataset. (Thus, DS1.LARGE is a temporal dataset generated from the catalog.xml document in the large DC/SD XBench dataset.) We use the notation DS#. to refer to a dataset generated with any of the classes.

Because of the design of our temporal data generation tool, which divides an input dataset into an initial slice and a “pool” of elements for use during the simulation, an input dataset must be large enough to accommodate the requested number of changes during the temporal simulation. This means that we could not use the SMALL XBench dataset to generate one thousand 1GB slices. Thus, depending on the characteristics of the defined temporal datasets, only certain input datasets make sense. We specify these below where applicable.

**Dataset 0 (DS0.**) This non-temporal dataset consists of the original, unmodified DC/SD XBench catalog.xml, along with the corresponding shredded non-temporal relations.

**Dataset 1 (DS1.**) This temporal dataset consists of DS0.* subjected to the temporal simulation with a weekly (7 day) time step and 700 changes per time step over a one year period. The initial slice of the temporal data consists of 10% of the elements from DS0.*; the remaining 90% are placed in a selection pool to be used by the simulation for the temporal changes. At each time step, the temporal changes are spread equally amongst the change types (insert, update, and delete) and temporal elements ($<$item$>$, $<$author$>$, $<$publisher$>$, and $<$related_item$>$). The total number of changes in the simulation is thus 36,400, over 52 time steps. We set the element selection type to “uniform” so that every element has an equal probability of being changed.
This dataset requires either the LARGE or HUGE XBench dataset as input.

**Dataset 2** (DS2.*) This temporal dataset is the same as DS1.*, except with a Gaussian element selection type. We make this choice to create a hot-spot of activity at the mean of the Gaussian distribution: elements with IDs near the mean will change frequently, while elements with IDs far from the mean will likely never change.

This dataset requires either the LARGE or HUGE XBench dataset as input.

**Dataset 3** (DS3.*) This temporal dataset is the same as DS2.*, except with a shorter time step (one day instead of seven days) and fewer changes per time step (100 versus 700). Since the number of required changes is the same, DS3.* will result in the same data characteristics as DS2.*, that is, 36,400 changes, except with seven times as many slices (364 versus 52).

This dataset requires either the LARGE or HUGE XBench dataset as input.

**Dataset 4** (DS4.*) This temporal dataset is created in order to provide a dataset with smaller slices and a substantive selection pool size to allow a large number of slices to be created. DS4.* is the same as DS1.*, except with only 0.4% of the elements from the input XBench dataset being selected for the initial slice (and thus 99.6% placed in the selection pool) and only 10 temporal changes per time step (Note that the time step is still 7 days). The required number of slices is set to 100 (or, equivalently, 1000 changes are required), although this parameter could easily be changed for evaluation purposes (for example, varying the number of slices to see the effect on the given application).

### 6.2 XBench Non-temporal XML Data

The dataset for the XBench benchmark is the non-temporal XML document of DS0.*.

### 6.3 τXBench Temporal XML Data

The datasets for the τXBench benchmark are the temporal XML documents of DS1.*, DS2.*, DS3.*, and DS4.*.

### 6.4 τXSchema Temporal XML Data

The datasets for the τXSchema benchmark are duplications of the τXBench datasets.
### Input Size

<table>
<thead>
<tr>
<th></th>
<th>DS0.SMALL</th>
<th>DS0.MEDIUM</th>
<th>DS0.LARGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total input size (catalog.xml)</td>
<td>11MB</td>
<td>104MB</td>
<td>1.1G</td>
</tr>
</tbody>
</table>

### Output Sizes

<table>
<thead>
<tr>
<th></th>
<th>DS0.SMALL</th>
<th>DS0.MEDIUM</th>
<th>DS0.LARGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>output.date.xml size (elements)</td>
<td>16.1MB (2500)</td>
<td>161MB (25000)</td>
<td>1.6GB (250000)</td>
</tr>
<tr>
<td>item.date.csv size (rows)</td>
<td>1.1MB (2500)</td>
<td>11MB (25000)</td>
<td>106MB (250000)</td>
</tr>
<tr>
<td>author.date.csv size (rows)</td>
<td>2.9MB (6264)</td>
<td>29MB (62497)</td>
<td>282MB (625057)</td>
</tr>
<tr>
<td>publisher.date.csv size (rows)</td>
<td>384KB (2500)</td>
<td>3.8MB (25000)</td>
<td>38MB (250000)</td>
</tr>
<tr>
<td>related_item.date.csv size (rows)</td>
<td>70K (6401)</td>
<td>803KB (62603)</td>
<td>9.1MB (624685)</td>
</tr>
<tr>
<td>item_author.date.csv size (rows)</td>
<td>70K (6264)</td>
<td>817KB (62497)</td>
<td>9.2MB (625057)</td>
</tr>
<tr>
<td>item_publisher.date.csv size (rows)</td>
<td>28K (2500)</td>
<td>321KB (25000)</td>
<td>3.7MB (250000)</td>
</tr>
<tr>
<td>Total output size</td>
<td>21MB</td>
<td>205MB</td>
<td>2.1GB</td>
</tr>
</tbody>
</table>

Table 3: The characteristics of DS0.*. DS0.HUGE is excluded since it is too large to run in the current implementation of τGenerator. Note that the sizes of the input catalog.xml files and the output output.date.xml files differ, even though they have the same content, because τGenerator outputs the XML files with indentation, whereas XBench output is more compressed.

### Input Parameters

<table>
<thead>
<tr>
<th></th>
<th>DS1.LARGE</th>
<th>DS2.LARGE</th>
<th>DS3.LARGE</th>
<th>DS4.LARGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL_PERCENTAGE</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>0.4</td>
</tr>
<tr>
<td>TIME_STEP</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Changes per time step¹</td>
<td>700</td>
<td>700</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>REQUIRED_CHANGES</td>
<td>36400</td>
<td>36400</td>
<td>36400</td>
<td>–</td>
</tr>
<tr>
<td>REQUIRED_SLICES</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>100</td>
</tr>
<tr>
<td>SELECTION_TYPE</td>
<td>uniform</td>
<td>gaussian</td>
<td>gaussian</td>
<td>uniform</td>
</tr>
<tr>
<td>SELECTION_STDDEV</td>
<td>–</td>
<td>100</td>
<td>100</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Output Sizes

<table>
<thead>
<tr>
<th></th>
<th>DS1.LARGE</th>
<th>DS2.LARGE</th>
<th>DS3.LARGE</th>
<th>DS4.LARGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>output.*.xml size (elements)</td>
<td>160MB (25000)</td>
<td>160MB (25000)</td>
<td>160MB (25000)</td>
<td>6.5MB (1000)</td>
</tr>
<tr>
<td>item.final.csv size (rows)</td>
<td>361MB (28306)</td>
<td>361MB (28306)</td>
<td>358MB (28200)</td>
<td>13.5MB (1100)</td>
</tr>
<tr>
<td>publisher.final.csv size (rows)</td>
<td>12.5MB (28306)</td>
<td>12.5MB (28306)</td>
<td>12.5MB (28200)</td>
<td>0.5MB (1100)</td>
</tr>
<tr>
<td>related_item.final.csv size (rows)</td>
<td>36.4MB (77236)</td>
<td>36.4MB (77236)</td>
<td>36.2MB (76723)</td>
<td>1.4MB (3010)</td>
</tr>
<tr>
<td>item_publisher.final.csv size (rows)</td>
<td>7MB (38224)</td>
<td>7MB (38224)</td>
<td>6MB (37800)</td>
<td>0.2MB (1297)</td>
</tr>
<tr>
<td>Total size</td>
<td>12.5GB</td>
<td>12.5GB</td>
<td>81GB</td>
<td>860MB</td>
</tr>
</tbody>
</table>

¹ Total number of changes to all temporal elements.

Table 4: The characteristics of the LARGE class of the four temporal datasets we define in τBench.
6.5 PSM Non-temporal Relations
The datasets for the PSM benchmark are the six non-temporal relations of DS0.*.

6.6 $\tau$PSM Temporal Relations
The datasets for the $\tau$PSM benchmark are the six temporal relations of DS1.*, DS2.*, DS3.*, and DS4.*.

6.7 Validation of Generated Data
The integrity of the datasets is crucial for the purposes of creating a benchmark. We therefore subject the generated datasets to a robust validation process.

The temporal XML document is validated in three ways:

- By the representational schema (created using the $\tau$XSchema tool suite [5]), to ensure it has the correct form.
- By the sequenced and non-sequenced constraints in the temporal schema (also created using the $\tau$XSchema tool suite), to ensure both types of constraints are ensured.
- By unsquashing [5] it and comparing each resulting slice to the generated slices with X-Diff [31] (see Section 8.3.4).

Should the temporal XML document pass all of these tests, we know that it is well formed and satisfies all of the constraints in the original XBench dataset.

The XML slices are validated by subjecting each to the original XBench schema using XMLLINT (see Section 8.3.1).

The temporal relations are validated by checking the primary key and referential integrity constraints of each relation (see Sections 8.3.2 and 8.3.3).

The relation slices are validated by checking the primary key and referential integrity constraints of each relation at each time period (see Sections 8.3.2 and 8.3.3).
7 Workloads

We now define the workloads of each benchmark and the relations between them. These conceptual relations are depicted in Figure 13. (Compare this with Figure 1 on page 2, in which the benchmarks themselves are in the same place but the transitions are somewhat different.) Figure 14 shows the process of creating each workload.

![Diagram of conceptual relationships between workloads](image)

Figure 13: Overview of the conceptual relationships between the τBench workloads.

7.1 XBench XQuery Queries

We employ the 20 original XBench XQuery queries, which were designed to subsume all of XQuery’s functionalities [1]. A description of each query and its highlighted functionality is shown in Table 5. Figure 15 shows an example query (Q2): “Find the title of the item which has matching author first name (Ben)”. In the query, `input()` is the input document (assumed to already be opened).

7.2 τXBench τXQuery Queries

What We Create We define a set of 100 τXQuery queries: a current and nonsequenced query for each of the original 20 XBench queries, and three validtime queries for each of the original 20 XBench queries: a short period validtime query that only considers 10% of the timeline, a medium period validtime query that considers half of the timeline, and a long period validtime query that considers 90% of the timeline.

How We Create It We add temporal statement modifiers to the XBench queries and use τXQuery to translate into XQuery. We add current for current queries and validtime for sequenced queries. Figure 16 shows a validtime query for XBench’s Q2. Non-sequenced queries are syntactically identical to the XBench queries.

How We Validate It Since we only add simple statement modifiers, no validation is required. At a different level though, if a way to evaluate τXQuery queries existed, we could evaluate each on a temporal document, then compare the result of that query with that of the analogous non-temporal query on the slices. This process would provide a validation of the evaluator, but also indirectly of the τXQuery queries themselves.

7.2.1 τXBench-Galax Queries

Galax does not support temporal statement modifiers, so it cannot directly execute the current nor sequenced τXBench queries. Gao has provided a method of source-to-source translation [25, 26]. We plan to manually map the τXQuery queries to conventional XQuery so that they can be executed by Galax.
Figure 14: Overview of mechanical generation of the \( \tau \)Bench workloads.

for $item in input()/catalog/:item
where $item/authors/author/name/first_name = "Ben"
return $item/title

Figure 15: Query Q2 of the XBench benchmark.

7.3 \( \tau \)XSchema \( \tau \)XQuery Queries

What We Create We create the same set of queries as the \( \tau \)XBench benchmark.

How We Create It The queries are duplicated from the \( \tau \)XBench queries. Note however that translation or evaluation may be able to benefit from the temporal constraints expressed in the temporal schema.

How We Validate It Since the queries are duplicated from the \( \tau \)XBench queries, once a translator that understands \( \tau \)XSchema is available, that translator could generate XQuery expressions. Validation of those expressions would compare their results to the mapped expressions from \( \tau \)XBench described in the previous section, using Galax.

7.4 PSM Queries

What We Create In creating the queries for PSM, we wanted to create a set that specifically highlighted the features of PSM, thus forming a micro-benchmark. Since there are fewer features in PSM than there are queries in the XBench benchmark, we omit six queries from the XBench benchmark. Some XBench queries are duplicated
Table 5: The features highlighted by the queries in the XBench Benchmark. Reproduced from [1].

<table>
<thead>
<tr>
<th>Description</th>
<th>Highlighted Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 Return the item that has matching item id attribute value (I1).</td>
<td>Top level exact match</td>
</tr>
<tr>
<td>Q2 Find the title of the item which has matching author first name (Ben).</td>
<td>Deep level exact match</td>
</tr>
<tr>
<td>Q3 Group items released in a certain year (1990), by publisher name and</td>
<td>Function application</td>
</tr>
<tr>
<td>calculate the total number of items for each group.</td>
<td></td>
</tr>
<tr>
<td>Q4 List the item id of the previous item of a matching item with id</td>
<td>Relative ordered access</td>
</tr>
<tr>
<td>attribute value (I2).</td>
<td></td>
</tr>
<tr>
<td>Q5 Return the information about the first author of item with a matching</td>
<td>Absolute ordered access</td>
</tr>
<tr>
<td>item id attribute value (I3).</td>
<td></td>
</tr>
<tr>
<td>Q6 Return item information where some authors are from certain country</td>
<td>Existential quantifier</td>
</tr>
<tr>
<td>(Canada).</td>
<td></td>
</tr>
<tr>
<td>Q7 Return item information where all its authors are from certain country</td>
<td>Universal quantifier</td>
</tr>
<tr>
<td>(Canada).</td>
<td></td>
</tr>
<tr>
<td>Q8 Return the publisher of an item with id attribute value (I4).</td>
<td>Regular path expressions (unknown</td>
</tr>
<tr>
<td></td>
<td>subpaths)</td>
</tr>
<tr>
<td>Q9 Return the ISBN of an item with id attribute value (I5).</td>
<td>Regular path expressions (unknown</td>
</tr>
<tr>
<td></td>
<td>element name)</td>
</tr>
<tr>
<td>Q10 List the item titles ordered alphabetically by publisher name, with</td>
<td>Sorting by string types</td>
</tr>
<tr>
<td>release date within a certain time period (from 1990-01-01 to 1995-01-01).</td>
<td></td>
</tr>
<tr>
<td>Q11 List the item titles in descending order by date of release with date</td>
<td>Sorting by non string types</td>
</tr>
<tr>
<td>of release within a certain time range (from 1990-01-01 to 1995-01-01).</td>
<td></td>
</tr>
<tr>
<td>Q12 Get the mailing address of the first author of certain item with id</td>
<td>Document structure preserving</td>
</tr>
<tr>
<td>attribute value (I6).</td>
<td>Missing elements</td>
</tr>
<tr>
<td>Q13 Return the names of publishers who publish books between a period of</td>
<td></td>
</tr>
<tr>
<td>time (from 1990-01-01 to 1991-01-01) but do not have FAX number.</td>
<td></td>
</tr>
<tr>
<td>Q17 Retrieve the item titles related by certain item with id attribute</td>
<td>Uni-gram search</td>
</tr>
<tr>
<td>value (I7).</td>
<td>References and joins</td>
</tr>
<tr>
<td>Q19 Retrieve the item title whose size (length<em>width</em>height) is bigger than</td>
<td>Datatype Cast</td>
</tr>
<tr>
<td>certain number (500000).</td>
<td></td>
</tr>
</tbody>
</table>

so that they can be implemented in a slightly different way to highlight a specific PSM feature. We thus define a total of 16 PSM queries.

Table 6 lists the queries and the features they highlight. The query set also contains one additional query (Q17b) not found in XBench, meant to be deliberately complicated and thus challenge query engines and language translators.

How We Create It  The queries for the PSM benchmark are manually derived by the second and third authors from the XQuery queries of the XBench benchmark. The authors began with the description of the XBench queries and manually implemented a semantically equivalent query in PSM.

Figure 17 shows query Q2 mapped from the original XBench query in Figure 15.

How We Validate It  We have developed a tool (see Section 8.3.5) that compares the output of running XQuery on an XBench query and running PSM on a PSM query. If the results are textually equivalent (after some modest reformatting), then we say that the queries are equivalent. We compare each pair of queries in this way (except Q17b, since it is new to the PSM benchmark) to determine the equivalence of the query sets.

7.4.1 PSM-DB2 Queries

We have translated the queries to conform to the DB2 syntax requirements, as depicted in Appendix D. Figure 18 shows Q2 after the translation.

7.5 τPSM Queries

What We Create  We define 80 τPSM queries that correspond to the 16 queries in the PSM benchmark: 48 sequenced queries (a short period, medium period, and long period sequenced query for each of the original 16), 16 non-sequenced, and 16 current.

How We Create It  The queries are derived directly from the PSM queries by adding simple temporal language modifiers (i.e., VALIDTIME for sequenced queries, NONSEQUENCED VALIDTIME for non-sequenced queries, and no change for current queries) to each of the queries. Figure 19 shows such a modified query Q2.
### Table 6: The features highlighted by the queries in the PSM Benchmark.

<table>
<thead>
<tr>
<th>Description</th>
<th>Highlighted Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2 Find the title of the item which has matching author first name (Ben).</td>
<td>SET with SELECT single row</td>
</tr>
<tr>
<td>Q2b Find the title of the item which has matching author first name (Ben).</td>
<td>Multiple SET statements</td>
</tr>
<tr>
<td>Q3 Group items released in a certain year (1990) by publisher name and calculated the total number of items for each group.</td>
<td>RETURN with SELECT single row</td>
</tr>
<tr>
<td>Q5 List the last name of the authors with a matching id of value (I3).</td>
<td>Function in the SELECT statement</td>
</tr>
<tr>
<td>Q6 Return item information where some authors are from certain country (Canada).</td>
<td>CASE statement</td>
</tr>
<tr>
<td>Q7 Return item information where all of its authors are from certain country (Canada).</td>
<td>WHILE statement</td>
</tr>
<tr>
<td>Q7b Return item information where all of its authors are from certain country (Canada).</td>
<td>REPEAT statement</td>
</tr>
<tr>
<td>Q8 Return the publisher of an item with id attribute value (I4).</td>
<td>Loop name with FOR statement</td>
</tr>
<tr>
<td>Q9 Return the ISBN of an item with id attribute value (I5).</td>
<td>CALL procedure from function</td>
</tr>
<tr>
<td>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).</td>
<td>IF without CURSOR</td>
</tr>
<tr>
<td>Q11 List the item titles in descending order by date of release within a certain time range (from 1990-01-01 to 1995-01-01).</td>
<td>Temporary table creation</td>
</tr>
<tr>
<td>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.</td>
<td>Local cursor declaration; FETCH, OPEN, and CLOSE statements</td>
</tr>
<tr>
<td>Q17 Return the first id of the items whose descriptions contain a certain word (&quot;hockey&quot;).</td>
<td>LEAVE statement</td>
</tr>
<tr>
<td>Q17b Return the first n ids of the items whose descriptions contain a certain word (&quot;hockey&quot;) and for which the number of pages is less than 500 if the author is Canadian. (n is an input parameter.) Also return the relevant date for each id. If all n are evaluated, results will be sorted by id.</td>
<td>Many (intentionally complicated)</td>
</tr>
<tr>
<td>Q19 Retrieve the item titles related by certain item with id attribute value (I7).</td>
<td>FOR statement</td>
</tr>
<tr>
<td>Q20 Retrieve the item title whose size (length<em>width</em>height) is bigger than certain number (500000).</td>
<td>ASSIGNMENT statement</td>
</tr>
</tbody>
</table>

---

-- Q2:  
-- Find the title of the item which has matching author 
-- first name (Ben).  
-- Feature:  
-- SET with SELECT single row

CREATE FUNCTION get_author_name(aid CHARACTER(10))  
RETURNS CHARACTER(50)  
READS SQL DATA  
LANGUAGE SQL  
BEGIN  
DECLARE fname CHARACTER(50);  
SET fname = (SELECT first_name FROM author WHERE author_id = aid);  
RETURN fname;  
END;

SELECT i.title  
FROM item i, item_author ia  
WHERE i.id = ia.item_id  
AND get_author_name(ia.author_id) = 'Ben';

Figure 17: Query Q2 of the PSM benchmark.
-- Q2:
-- Find the title of the item which has matching author
-- first name (Ben).
-- Feature:
-- SET with SELECT single row

CREATE FUNCTION db2_orig_get_author_name(aid CHARACTER(10))
RETURNS CHARACTER(50)
NO EXTERNAL ACTION
BEGIN
DECLARE fname CHARACTER(50);
SET fname = (SELECT first_name FROM author WHERE author_id = aid);
RETURN fname;
END;

SELECT i.title
FROM item i, item_author ia
WHERE i.id = ia.item_id
AND db2_orig_get_author_name(ia.author_id) = 'Ben';

Figure 18: Query Q2 of the PSM-DB2 benchmark.

CREATE FUNCTION get_author_name(aid CHARACTER(10))
RETURNS CHARACTER(50)
READS SQL DATA
LANGUAGE SQL
BEGIN
DECLARE fname CHARACTER(50);
SET fname = (SELECT first_name FROM author WHERE author_id = aid);
RETURN fname;
END;

VALIDTIME SELECT i.title
FROM item i, item_author ia
WHERE i.id = ia.item_id
AND get_author_name(ia.author_id) = 'Ben';

Figure 19: Query Q2 of the ρPSM benchmark.
CREATE FUNCTION db2_orig_get_author_name(aid CHARACTER(10)) RETURNS CHARACTER(50) NO EXTERNAL ACTION F1: BEGIN DECLARE fname CHARACTER(50); SET fname = (SELECT first_name FROM author WHERE author_id = aid); RETURN fname; END VALIDTIME SELECT i.title FROM item i, item_author ia WHERE i.id = ia.item_id AND db2_orig_get_author_name(ia.author_id) = 'Ben';

Figure 20: Query Q2 of the τPSM-DB2 benchmark.

How We Validate It Since we only add statement modifiers, no validation is required.

7.5.1 τPSM-DB2 Queries

We have translated the queries to conform to the DB2 syntax requirements, as depicted in Appendix D. Figure 20 shows the sequenced Q2 after the translation.
8 Supporting Tool Suite

To facilitate the creation and validation of the τBench suite of benchmarks, we have developed a suite of tools that generate temporal data, validate the data in various ways, and validate the correctness of the queries.

The tool suite is distributed according the terms of the GNU Lesser General Public License [32]. It can be obtained on the TimeCenter web page [11].

8.1 τGenerator: Simulating Temporal Data

In this section, we introduce and describe τGenerator, a tool that creates temporal data from the dataset generated by XBench. We describe the simulation method used to τGenerator to incrementally change the data, and then describe the inputs and outputs of the simulation. We conclude with a brief overview of the compilation and usage of τBench.

To create the temporal data, we implement a simulation that consists of a user-specified number of time steps along with a user-specified number of changes to the data within each time step. At each time step, a subset of the elements from the original document are changed, creating a new slice of the temporal document. The simulation continues until all of the required changes are made.

8.1.1 Simulation Description

We populate the four temporal elements by randomly changing the values of the elements’ text content and sub-elements at specified time intervals (i.e., a time-based simulation). The simulation consists of the following four steps.

1. **Setup.** We first read the user-created parameters file to initialize the simulation parameters. (See Section 8.1.3 for more information about the input parameters and their required format.) We then use DOM [33] to load the user-specified XBench document (catalog.xml) into memory.

   We then create temporal items out of our chosen temporal elements <item>, <author>, <publisher>, and <related_items>). We do so by creating a temporal element with a single version. For example, for each original <author> element in catalog.xml, we create an <author_repItem> element with a single sub-element named <author_version>, which itself contains the original <author> element as a sub-element.

2. **Initial snapshot selection.** We select a user-specified percentage of <item> temporal elements (and all of their sub-elements) from catalog.xml for the initial slice. We select the elements uniformly at random from catalog.xml. Elements that are not selected for the initial slice are put into a pool for later use.

![Figure 21: Overview of the τGenerator process.](image-url)
We set the begin and end time of the first (and only) version of each selected temporal element to the user-specified begin date and the user-specified forever date, respectively.

3. **Time step loop.** We increase the current time by the user-specified time step.

   (a) **Select elements to change.** We first select the temporal elements that are currently active (i.e., the “end” attribute of the element’s latest version is equal to or greater than the current time). From this set of active temporal elements, we randomly select a user-specified number of elements to change. The elements are selected randomly based on their ID values, according to the user-specified selection type.

   • If the selection type is “uniform”, then each element has an equal chance of being selected for change.
   • If the selection type is “gaussian”, then elements whose ID values are close to the mean of the user-specified Gaussian distribution (Table 7) are more likely to be selected; elements whose ID values are far from the mean are less likely to be selected.

   We do not select the same ID value more than once in any given time step.

   (b) **Change the selected elements.** Once a subset of the active temporal elements has been selected for change, we delete, insert, and update a user-specified number of the selected elements with the following methodology.

   • When the change type is “delete”, we set the end time of the current version (and all temporal sub-elements) to the current simulation time.
   • When the change type is “insert”, we create a new temporal element with a single version, set the begin time of the version to the current simulation time, set the end time of the version to the user-specified forever time, and set the actual element values to those of a randomly selected element from the pool.
   • When the change type is “update”, we set the end time of the current version (and all of its temporal sub-elements) to the current simulation time, create a new copy of the current version with begin time set to the current simulation time, and modify a randomly-selected sub-element of the new version by copying the data from a randomly-selected element from the pool. (Note that only a single value in the temporal element’s tree will be modified. For example, if an <author> element is selected for update, then perhaps its <first_name> or <address> sub-elements will change, but not both.)

   When an element is taken from the pool, it is selected according the same user-specified selection type as in Step 3 (a). Elements taken from the pool (for either the insert or update steps) are not replaced.

   (c) **Output slices in XML and CSV.** If the user-specified slice output parameter is set, we output the current slice in XML and CSV formats. See Section 8.1.4 for more details about the output files.

   (d) **Check stop criteria.** If we have exceeded the specified number of necessary changes, we exit the time step loop and continue to Step 4. Otherwise, we return to Step 3.

4. **Output temporal documents.** Finally, we output the final temporal documents in XML and CSV format. See Section 8.1.4 for more details about the output files.

### 8.1.2 Maintaining the Original Schema Constraints

During the above simulation process, it is important to maintain the original constraints defined in the XBench XML schema. This will ensure that the generated data validates to the sequenced semantics of the original schema. In addition, we add additional checks in the simulation to maintain the set of non-sequenced constraints in the τXSchema benchmark.

We need not consider the original data type constraints, because the simulation does not change the type of data at any point. Similarly, we need not consider identity constraints (which ensure, for example, that the <item>’s id attribute is globally unique) because the simulation does not generate any ids of its own—all <item>s keep their originally-assigned ids. Thus, as long as the original data is correct, the generated data will also be correct.
However, we do need to consider the referential integrity constraint between \textit{item} ids and \textit{related items}. Since only a subset of \textit{item} elements are active at any given time, there is no guarantee that \textit{related items} will point to current active \textit{item} ids. Thus, the simulation must provide this guarantee explicitly. The simulation achieves this by adding a check at the end of each time-step. The check builds a list of all currently active \textit{item} ids. Then, for any currently active \textit{related items} that refer to an \textit{item} id not in the list, the simulation replaces the value with an id in the list. Note that this ensures both sequenced and non-sequenced referential integrity constraints between \textit{item} ids and \textit{related items}.

We also need to consider the cardinality constraints placed on the \textit{author} element. Since the simulation has the ability to add new \textit{author}s and delete existing \textit{author}s at any point in time, the simulation must be careful not to violate the maxOccurs and minOccurs constraints. This is true for both the sequenced (i.e., at any given time, \textit{item} must have between 1 and 4 \textit{author}s) and non-sequenced (i.e., in any given year, an \textit{item} can have up to 6 \textit{author}s) variants of this constraint. The simulation achieves these by maintaining an author counter for each \textit{item} and each time period. For the sequenced constraint, the counter is incremented and decremented as the simulation adds or removes \textit{author}s. For the non-sequenced, the counter is only incremented when an \textit{author} is inserted. Finally, when an \textit{item} is selected by the simulation to insert or remove an \textit{author}, the simulation first consults this counter; if the counter indicates that such an action will violate a constraint, the simulation chooses another \textit{item} to change. The simulation will continue trying to choose a viable author for a predefined number of iterations, and then will halt to avoid infinite loops.

Thus, the resulting temporal data is made consistent for both sequenced and non-sequenced constraints.

### 8.1.3 Input

Table 7 lists the names of the input parameters for $\tau$Generator along with a brief description of each. The user places the input parameters and their desired values into a \textit{parameters} file, which is a conventional text file. The parameter file should have one \textit{parameter-value pair} on each line. A parameter-value pair is a line containing the name of a parameter, some white space, and the desired value of the parameter. For example, to set the value of the \textit{TIME_STEP} parameter to 15 days, then the following line should appear somewhere the parameters file.

\texttt{TIME_STEP 15}

Note that order is not important in the parameters file, and if a parameter is listed more than once, then the last instance will be used. If a parameter is not listed in the parameters file, then the parameter takes the default value specified in Table 7. Comment lines are also allowed, which are preceded with the ‘#’ character:

\# This is a comment line.

Blank lines and extra white space are allowed:

\# This is line 1.

\# This is line 3.

An example parameters file is listed in Appendix C.

### 8.1.4 Output

$\tau$Generator has the ability to output a temporal XML document and corresponding shredded relations into a user-specified output directory. If the user-specified slice output parameters are set, then at each time step, $\tau$Generator will also output the current version of each of the six files.

The output files are summarized in Table 8.

**Shredding XML Data into Relations**  
Shredding is the process of transforming an XML tree into a set of two dimensional DBMS relations [34]. Doing so often requires generating additional tables that contain relational or hierarchical information. For example, an additional table is required to list the relations between \textit{item}s and \textit{author}s in the XBench DC/SD \texttt{catalog.xml} file. Note that no information is lost during the shredding process.
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT_FILE</td>
<td>The string name of the input file (produced by XBench).</td>
<td>catalog.xml</td>
</tr>
<tr>
<td>OUTPUT_DIR</td>
<td>The string name of the directory in which to place the generated output files.</td>
<td>output</td>
</tr>
<tr>
<td>OUTPUT_XML</td>
<td>A flag (0 or 1) indicating if the simulation should output the temporal XML file when the simulation ends.</td>
<td>1</td>
</tr>
<tr>
<td>OUTPUT_RELATIONS</td>
<td>A flag (0 or 1) indicating if the simulation should output the shredded relations when the simulation ends.</td>
<td>0</td>
</tr>
<tr>
<td>OUTPUT_XML_SLICES</td>
<td>A flag (0 or 1) indicating if the simulation should output the XML slice at each time step.</td>
<td>0</td>
</tr>
<tr>
<td>OUTPUT_RELATION_SLICES</td>
<td>A flag (0 or 1) indicating if the simulation should output the shredded relations at each time step.</td>
<td>0</td>
</tr>
<tr>
<td>INITIAL_PERCENTAGE</td>
<td>The positive floating-point percentage of elements in INPUT_FILE to be placed in the initial slice.</td>
<td>10</td>
</tr>
<tr>
<td>START_TIME</td>
<td>The starting date of the simulation.</td>
<td>2010-01-01</td>
</tr>
<tr>
<td>FOREVER_TIME</td>
<td>The date of &quot;forever,&quot; a time not reachable.</td>
<td>2099-12-31</td>
</tr>
<tr>
<td>TIME_STEP</td>
<td>The number of days between time steps.</td>
<td>7</td>
</tr>
<tr>
<td>REQUIRED_CHANGES</td>
<td>The positive integer number of changes before the simulation terminates.</td>
<td>100</td>
</tr>
<tr>
<td>ITEM_TIME_STEP_INSERTS</td>
<td>The non-negative integer number of &lt;item&gt;s that are inserted at each time step.</td>
<td>1</td>
</tr>
<tr>
<td>ITEM_TIME_STEP_DELETES</td>
<td>The non-negative integer number of &lt;item&gt;s that are deleted at each time step.</td>
<td>1</td>
</tr>
<tr>
<td>ITEM_TIME_STEP_UPDATES</td>
<td>The non-negative integer number of &lt;item&gt;s that are updated at each time step.</td>
<td>1</td>
</tr>
<tr>
<td>AUTHOR_TIME_STEP_INSERTS</td>
<td>The non-negative integer number of &lt;author&gt;s that are inserted at each time step.</td>
<td>1</td>
</tr>
<tr>
<td>AUTHOR_TIME_STEP_DELETES</td>
<td>The non-negative integer number of &lt;author&gt;s that are deleted at each time step.</td>
<td>1</td>
</tr>
<tr>
<td>PUBLISHER_TIME_STEP_DELTIES_INSERTS</td>
<td>The non-negative integer number of &lt;publisher&gt;s that are deleted/inserted at each time step.</td>
<td>1</td>
</tr>
<tr>
<td>RELATION_TIME_STEP_DELTIES_INSERTS</td>
<td>The non-negative integer number of &lt;related_item&gt;s that are deleted/inserted at each time step.</td>
<td>1</td>
</tr>
<tr>
<td>SELECTION_TYPE</td>
<td>The string name of the distribution for selecting elements to change; options are gaussian or uniform.</td>
<td>uniform</td>
</tr>
<tr>
<td>SELECTION_STDDEV</td>
<td>For SELECTION_TYPE of gaussian, the floating-point positive standard deviation of the distribution. (The mean is set by the simulation to half of the number of elements initially selected.)</td>
<td>10.0</td>
</tr>
<tr>
<td>NULL_VALUE</td>
<td>The string value to use for NULL columns when shredding the XML documents into CSV.</td>
<td>null</td>
</tr>
</tbody>
</table>

Table 7: The inputs into τGenerator.

We shred the time-varying catalog.xml into six relations: four for the original four temporal elements, one to map author IDs to item IDs, and one to map publisher IDs to item IDs. In general, each column of each shredded relation corresponds to the text content of a subelement of the corresponding temporal element. For example, the (simplified) <author> temporal element (note: not actually generated by τGenerator) with two versions shown in Figure 22 would be shredded into the two tuples shown in Table 9.

Appendix B.1 contains the relational schemas for the six shredded relations.

8.1.5 Compilation and Usage

τGenerator has been made publicly available on the TimeCenter website [11]. The Xerces [35] library is also required and comes bundled with the τGenerator distribution.

τGenerator ships with the following files.

- README.txt. A helper file that indicates how to compile and run τGenerator.
- tGenerator.java. The single source code file.
- catalog.xml. A sample bookstore catalog generated by XBench (10MB).
- xerces-2.2.1.jar. The Xerces distribution.
Once downloaded, τGenerator can be compiled with the following command.

```java
javac tGenerator.java -cp ./xerces-2.2.1.jar
```

Or simply:

```bash
compile
```

This will create a Java class file named `tGenerator.class` in the current directory.

τBench can be executed with the following command.

```bash
java -cp ./xerces-2.2.1.jar: . tGenerator parameters.txt
```

The format of the `parameters.txt` file is given in Section 8.1.3. For larger scenarios of XBench, the memory of the Java Virtual Machine (JVM) needs to be increased. The following command, for example, will provide the JVM with 28 gigabytes of memory.

```bash
java -Xmx28g -cp ./xerces-2.2.1.jar: . tGenerator parameters.txt
```

We have found that running τGenerator with XBench’s small (10MB) and normal (100MB) class sizes requires less than 2GB of memory while the large (1GB) class requires 28GB of memory. We have not yet tested the huge (10GB) class.

A sample `catalog.xml` and parameters file is bundled with the τGenerator distribution.

<table>
<thead>
<tr>
<th>first_name</th>
<th>date_of_birth</th>
<th>name_of_city</th>
<th>...</th>
<th>begin</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tandy</td>
<td>1972-06-29</td>
<td>Oakville</td>
<td></td>
<td>1992-12-04</td>
<td>2004-08-05</td>
</tr>
<tr>
<td>Tandy</td>
<td>1972-06-29</td>
<td>Lincoln</td>
<td></td>
<td>2004-08-05</td>
<td>2005-05-01</td>
</tr>
</tbody>
</table>

Table 9: The two versions of the `<author>` temporal element, shredded into two tuples.
8.2 τCorruptor: Corrupting Temporal Data

Most experienced developers know that it is useful to manually cause an application to fail during the testing phase of the application, to exercise the error checking pieces of the application. The same is true for data generation: it is useful to corrupt the generated data in a testing phase to determine if the schemas and validators can detect the corruptions.

We have developed a tool, called τCorruptor, for just this purpose. τCorruptor takes as input a temporal XML document and outputs another temporal XML document which is identical as the input document except for one victim element. The victim element is changed to directly contradict one of the constraints on the data.

τCorruptor has proven useful in the development phases of τGenerator as well as for the development phase of τXMLLint.

8.2.1 Input

The input into τCorruptor is a temporal XML document generated by τGenerator, as well as an integer code specifying which of the seven temporal constraints defined in the τXSchema benchmark schemas to invalidate. The codes have the following mapping.

1. Sequenced cardinality. Duplicates an entire <author> four times (but keeps each author_id unique).
2. Sequenced referential integrity. Changes a <related_item>’s item_id to a random string.
4. Sequenced datatype. Changes the <number_of_pages> to a random string.
5. Non-sequenced identity. Copies the id of an <item> at time \( t_1 \) to the id of an <item> at time \( t_2 \), where \( t_1 \neq t_2 \).
6. Non-sequenced referential integrity. Changes a <related_item>’s item_id to a random string.
7. Non-sequenced cardinality. Duplicates an entire <author>RepItem> six times (but keeps each author_id unique).

8.2.2 Output

τCorruptor will randomly select a victim node and apply the changes specified by the input. It will then output the modified temporal XML document.

8.2.3 Compilation and Usage

τCorruptor can be compiled with the following command.

```
javac tCorruptor.java -cp ./xerces-2.2.1.jar
```

Or simply:

```
compile
```

This will create a Java class file named τCorruptor.class in the current directory.

τBench can be executed with the following command.

```
java -cp ./xerces-2.2.1.jar: . τCorruptor input.xml actionCode
```

8.3 Validating the Correctness of Schemas, Data and Workloads

A critical aspect of the τBench suite of benchmarks is ensuring that the generated data, schema, and workloads are correct, because otherwise the benchmarks would be meaningless. Ascertaining the correctness of the benchmarks, however, has proven to be a difficult task due to the complexities involved. To aid us in this task, we have developed a set of tools and processes to automate the validation process, which we describe in this section.

We now describe our tools and processes for schemas, data, and workloads.
8.3.1 Validating the XML Output (validateAgainstSchemas.pl)

This Perl script validates each generated XML slice against its conventional XML schema (DCSD.xsd) using XMLLINT. It also validates the generated temporal document against its temporal schema using τXMLLINT.

These checks ensure that each generated slice is well formed and consistent with the schema constraints, and that the generated temporal document is well formed and consistent with the sequenced semantics of the conventional schema.

8.3.2 Checking Primary Keys of Relations (checkKeys.pl)

This Perl script checks each shredded relation to ensure that each primary key is unique across the relation. It so does by building a hash table of primary keys in each table. If a new row is encountered whose key is already defined in the hash table, then the script reports an error.

Tables whose primary key are defined across multiple columns require multi-dimensional hash tables.

8.3.3 Checking Referential Integrity of Relations (checkRefs.pl)

This Perl script checks each shredded relation to ensure that each referential integrity constraint is valid across the relation. It does so by first reading the entire relation and creating a hash table of the referenced primary keys. A second pass ensures that each value in a foreign key column is defined in the hash table.

8.3.4 Comparing XML Slices to the Temporal XML Document (checkSlices.pl)

This Perl script ensures that the generated temporal XML document and the corresponding XML slices are equivalent. The script makes use of UNSQUASH, which is part of the τXSchemas tool suite. UNSQUASH takes as input a temporal XML document and extracts each slice. The script then compares each generated slice with each extracted slice using X-Diff [31] to ensure that they are equivalent. If all the slices are equivalent, then the temporal XML document and generated slices are equivalent.

8.3.5 Comparing XQuery and PSM Queries (compareQueries.pl)

This Perl script performs two queries and compares their output. First, it runs an XQuery query on the non-temporal XML data and saves the output. Then, it runs the corresponding PSM query on the shredded relational data and saves the output. Since the output formats are slightly different, the script reformats the PSM output to be of the same format as the XQuery output. Then, the script can compare the query results. If the two results are textually identical, then the queries achieved the same functionality.

If the script succeeds for all queries, then the two query sets are identical.
9 Conclusions and Future Research

In this report, we have introduced τBench, a suite of benchmarks based upon the XBench benchmark. The τBench suite is comprised of both non-temporal and temporal datasets generated by multiple data generation methodologies, spans a range of database technologies and tools, and contains several workload paradigms.

The τBench benchmark suite currently consists of five benchmarks: XBench, τXBench, τXS, PSM, and τPSM, and three DBMS-specific implementations, τXBench-Galax, PSM-DB2, and τPSM-DB2, each consisting of schemas, datasets, and workloads. The τBench tool suite, which was built to aid the construction and validation of the benchmark suite, consists of a temporal data generation tool (τGenerator), a data corruption tool for testing purposes (τCorruptor), and a set of scripts that validate the generated data in various ways. Both the benchmark suite and tool suite are available online [11].

During the creation of τBench, we had five primary goals:

• Construct a framework for creating new benchmarks from existing components.
• Rely on statement modifiers when possible, easing the transformation from the non-temporal domain into the temporal.
• Construct robust and correct benchmarks that are logically consistent and syntactically valid. We achieved this robustness and validity through a triangulation of tests against the data, schemas, and workloads that tested the entire benchmark from a multitude of angles.
• Define benchmark components that were independent of a specific DBMS, then derive DBMS-specific components via a small number of transformations.
• Use a standard benchmark as the foundation for new benchmarks, thereby ensuring the entire suite is solid and useful.

The resulting τBench suite is an ecosystem of related benchmarks. In this ecosystem, components of existing benchmarks flow into the construction of new benchmarks, all the while being validated by several measures. As each new benchmark is constructed and validated, the foundational benchmarks gain strength because their components are again being validated, this time in a new light. And while all the benchmarks are tightly interconnected (in that they are derived from each other and share data, schemas, and workloads), their use in research and industry is completely independent (in that any single benchmark can be used without any of the others).

Future work for τBench includes defining additional benchmarks for new and existing database technologies, and extending the datasets to include other categories of XBench datasets. We also want to use suitable language-to-language translators when they become available to create executable workloads for τXBench-Galax and τPSM-DB2, to enable validation of these workloads.

Future work for τGenerator includes:

• Addressing the inefficiency of DOM. The execution time of some routines in τGenerator could be vastly improved with a better use of DOM iterators. Also, the memory requirements of τGenerator should be addressed by reusing data structures where possible.
• Porting to τDOM [36]. τDOM contains temporal support for manipulating DOM objects in memory. Much of the internal complexities of τGenerator could be reduced with the use of τDOM.
• Considering other output categories of XBench. Currently, τGenerator is built on the DC/SD category of XBench output, but the other three may also be useful. This would involve generalizing the internals of τGenerator.
References


A XML Schemas

A.1 Conventional XML Schema (DCSD.xsd)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="FAX_number" type="xs:string"/>
  <xs:element name="ISBN" type="xs:string"/>
  <xs:element name="attributes">
    <xs:complexType>
      <xs:all>
        <xs:element ref="ISBN"/>
        <xs:element ref="number_of_pages"/>
        <xs:element ref="type_of_book"/>
        <xs:element ref="size_of_book"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="author">
    <xs:complexType>
      <xs:all>
        <xs:element name="name">
          <xs:complexType>
            <xs:all>
              <xs:element ref="first_name"/>
              <xs:element ref="middle_name"/>
              <xs:element ref="last_name"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
        <xs:element ref="date_of_birth"/>
        <xs:element ref="biography"/>
        <xs:element name="contact_information">
          <xs:complexType>
            <xs:all>
              <xs:element name="mailing_address">
                <xs:complexType>
                  <xs:all>
                    <xs:element ref="street_information"/>
                    <xs:element ref="name_of_city"/>
                    <xs:element ref="name_of_state"/>
                    <xs:element ref="zip_code"/>
                    <xs:element name="name_of_country" type="xs:string"/>
                  </xs:all>
                </xs:complexType>
              </xs:element>
              <xs:element ref="phone_number"/>
              <xs:element ref="email_address"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
        <xs:attribute name="author_id" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="authors">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="author" maxOccurs="4"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="catalog">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="item" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- Sequenced Identify Constraint: -->
  <!-- Item ISBNs are unique. -->
  <xs:unique name="ISBNUnique">
    <xs:selector xpath=".//item/attributes"/>
    <xs:field xpath="ISBN"/>
  </xs:unique>
  <!-- Sequenced Referential Integrity: -->
  <!-- A related item should refer to a valid item -->
</xs:schema>
```
<xs:complexType>
  <xs:all>
    <xs:element ref="thumbnail"/>
    <xs:element ref="image"/>
  </xs:all>
</xs:complexType>
</xs:element>
<xs:element name="middle_name" type="xs:string"/>
<xs:element name="name_of_city" type="xs:string"/>
<xs:element name="name_of_country" type="xs:string"/>
<xs:element name="name_of_state" type="xs:string"/>
<!-- Sequenced Datatype: -->
<!-- The number_of_pages must be of type short. -->
<xs:element name="number_of_pages" type="xs:short"/>
<xs:element name="phone_number" type="xs:string"/>
<xs:complexType>
  <xs:all>
    <xs:element ref="suggested_retail_price"/>
    <xs:element ref="cost"/>
    <xs:element ref="when_is_available"/>
    <xs:element ref="quantity_in_stock"/>
  </xs:all>
</xs:complexType>
</xs:element>
<xs:element name="publisher"/>
<xs:complexType>
  <xs:all>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="contact_information">
      <xs:complexType>
        <xs:all>
          <xs:element ref="mailing_address">
            <xs:complexType>
              <xs:all>
                <xs:element ref="street_information"/>
                <xs:element ref="name_of_city"/>
                <xs:element ref="name_of_state"/>
                <xs:element ref="zip_code"/>
                <xs:element ref="country"/>
              </xs:all>
            </xs:complexType>
          </xs:element>
        </xs:all>
      </xs:complexType>
    </xs:element>
    <xs:element ref="FAX_number" minOccurs="0"/>
    <xs:element ref="phone_number"/>
    <xs:element ref="web_site"/>
  </xs:all>
</xs:complexType>
</xs:element>
<xs:element name="quantity_in_stock" type="xs:byte"/>
<xs:element name="related_items">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="related_item" minOccurs="0" maxOccurs="5"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="related_items" type="xs:string"/>
<xs:complexType>
  <xs:all>
    <xs:element ref="item_id"/>
  </xs:all>
</xs:complexType>
</xs:element>
<xs:element name="related_items" type="xs:string"/>
<xs:complexType>
  <xs:all>
    <xs:element ref="item_id" minOccurs="0" maxOccurs="5"/>
  </xs:all>
</xs:complexType>
</xs:element>
<xs:element name="size_of_book" type="xs:string"/>
<xs:complexType>
  <xs:all>
    <xs:element ref="length"/>
    <xs:element ref="width"/>
    <xs:element ref="height"/>
  </xs:all>
</xs:complexType>
</xs:element>
<xs:element name="street_address" type="xs:string"/>
<xs:element name="street_information" type="xs:string"/>
<xs:complexType>
<xs:element ref="street_address" maxOccurs="2"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="subject" type="xs:string"/>
<xs:element name="suggested_retail_price">
<xs:simpleContent>
<xs:extension base="xs:decimal">
<xs:attribute name="currency" type="xs:string" use="required"/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>
<xs:element name="thumbnail">
<xs:complexType>
<xs:all>
<xs:element ref="data"/>
</xs:all>
</xs:complexType>
</xs:element>
<xs:element name="title" type="xs:string"/>
<xs:element name="type_of_book" type="xs:string"/>
<xs:element name="when_is_available" type="xs:date"/>
<xs:element name="width">
<xs:simpleContent>
<xs:extension base="xs:decimal">
<xs:attribute name="unit" type="xs:string" use="required"/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>
<xs:element name="zip_code" type="xs:string"/>
</xs:schema>
A.2 Temporal XML Schema

```xml
<?xml version="1.0" encoding="utf-8"?>
<temporalSchema
xmlns="http://www.cs.arizona.edu/tau/TSSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cs.arizona.edu/tau/TSSchema/etc/TSSchema.xsd">
    <conventionalSchema>
        <include schemaLocation="DCSD.xsd" />
    </conventionalSchema>
    <annotationSet>
        <logical>
            <item target="catalog/item" />
            <item target="catalog/item/authors/author" />
            <item target="catalog/item/publisher" />
            <item target="catalog/item/related_items" />
        </logical>
        <physical>
            <default>
                <format plugin="XMLSchema" granularity="days"/>
            </default>
            <!-- default timestamps are at logical items -->
        </physical>
    </annotationSet>
</temporalSchema>
```
A.3 Conventional XML Schema with Added Constraints

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="FAX_number" type="xs:string"/>
  <xs:element name="ISBN" type="xs:string"/>
  <xs:element name="attributes">
    <xs:complexType>
      <xs:all>
        <xs:element ref="ISBN"/>
        <xs:element ref="number_of_pages"/>
        <xs:element ref="type_of_book"/>
        <xs:element ref="size_of_book"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="author">
    <xs:complexType>
      <xs:all>
        <xs:element name="name">
          <xs:complexType>
            <xs:all>
              <xs:element ref="first_name"/>
              <xs:element ref="middle_name"/>
              <xs:element ref="last_name"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
        <xs:element ref="date_of_birth"/>
        <xs:element ref="biography"/>
        <xs:element name="contact_information">
          <xs:complexType>
            <xs:all>
              <xs:element name="mailing_address">
                <xs:complexType>
                  <xs:all>
                    <xs:element ref="street_information"/>
                    <xs:element ref="name_of_city"/>
                    <xs:element ref="name_of_state"/>
                    <xs:element ref="zip_code"/>
                    <xs:element name="name_of_country" type="xs:string"/>
                  </xs:all>
                </xs:complexType>
              </xs:element>
              <xs:element ref="phone_number"/>
              <xs:element ref="email_address"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
        <xs:attribute name="author_id" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="authors">
    <xs:complexType>
      <xs:sequence>
        <!-- Sequenced Cardinality: -->
        <!-- An item must have between 1 and 4 authors. -->
        <xs:element ref="author" maxOccurs="4"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="biography" type="xs:string"/>
  <xs:element name="catalog">
    <xs:complexType>
      <xs:sequence>
        <!-- Sequenced Identify Constraint: -->
        <!-- Item ISBNs are unique. -->
        <xs:unique name="ISBNUnique">
          <xs:selector xpath="./item/attributes"/>
          <xs:field xpath="ISBN"/>
          <xs:unique/>
          <!-- Sequenced Referential Integrity: -->
          <!-- A related item should refer to a valid item -->
          <xs:key name="itemID"/>
          <xs:selector xpath="./item"/>
          <xs:field xpath="@id"/>
        </xs:unique>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
<xs:element name="middle_name" type="xs:string"/>
<xs:element name="name_of_city" type="xs:string"/>
<xs:element name="name_of_country" type="xs:string"/>
<xs:element name="name_of_state" type="xs:string"/>
<!-- Sequenced Datatype: -->
<!-- The number_of_pages must be of type short. -->
<xs:element name="number_of_pages" type="xs:short"/>
<xs:element name="phone_number" type="xs:string"/>
<xs:element name="pricing">
  <xs:complexType>
    <xs:all>
      <xs:element ref="suggested_retail_price"/>
      <xs:element ref="cost"/>
      <xs:element ref="when_is_available"/>
      <xs:element ref="quantity_in_stock"/>
    </xs:all>
  </xs:complexType>
</xs:element>
<xs:element name="publisher">
  <xs:complexType>
    <xs:all>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="contact_information">
        <xs:complexType>
          <xs:all>
            <xs:element name="mailing_address">
              <xs:complexType>
                <xs:all>
                  <xs:element ref="street_information"/>
                  <xs:element ref="name_of_city"/>
                  <xs:element ref="name_of_state"/>
                  <xs:element ref="zip_code"/>
                  <xs:element ref="country"/>
                </xs:all>
              </xs:complexType>
            </xs:element>
            <xs:element ref="FAX_number" minOccurs="0"/>
            <xs:element ref="phone_number"/>
            <xs:element ref="web_site"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:all>
  </xs:complexType>
</xs:element>
<xs:element name="quantity_in_stock" type="xs:byte"/>
<xs:element name="related_item">
  <xs:complexType>
    <xs:all>
      <xs:element ref="item_id"/>
    </xs:all>
  </xs:complexType>
</xs:element>
<xs:element name="related_items">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="related_item" minOccurs="0" maxOccurs="5"/>
    </xs:sequence>
    <xs:attribute name="related_items_id" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:element name="size_of_book">
  <xs:complexType>
    <xs:all>
      <xs:element ref="length"/>
      <xs:element ref="width"/>
      <xs:element ref="height"/>
    </xs:all>
  </xs:complexType>
</xs:element>
<xs:element name="street_address" type="xs:string"/>
<xs:element name="street_information">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="street_address" maxOccurs="2"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType>
  <xs:element name="subject" type="xs:string"/>
  <xs:element name="suggested_retail_price">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:decimal">
          <xs:attribute name="currency" type="xs:string" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="thumbnail">
    <xs:complexType>
      <xs:all>
        <xs:element ref="data"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="title" type="xs:string"/>
  <xs:element name="type_of_book" type="xs:string"/>
  <xs:element name="when_is_available" type="xs:date"/>
  <xs:element name="width">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:decimal">
          <xs:attribute name="unit" type="xs:string" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="zip_code" type="xs:string"/>
</xs:schema>
A.4 Representational XML Schema

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

<xs:element name="catalog_RepItem">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="catalog_Version" minOccurs="1" maxOccurs="unbounded"/>
            <xs:attribute name="isItem" type="xs:string" use="required"/>
            <xs:attribute name="originalElement" type="xs:string" use="required"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="catalog_Version">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="catalog"/>
            <xs:attribute name="begin" type="xs:date" use="required"/>
            <xs:attribute name="end" type="xs:date" use="required"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="item_RepItem">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="item_Version" minOccurs="1" maxOccurs="unbounded"/>
            <xs:attribute name="isItem" type="xs:string" use="required"/>
            <xs:attribute name="originalElement" type="xs:string" use="required"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="item_Version">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="item"/>
            <xs:attribute name="begin" type="xs:date" use="required"/>
            <xs:attribute name="end" type="xs:date" use="required"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="author_RepItem">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="author_Version" minOccurs="1" maxOccurs="unbounded"/>
            <xs:attribute name="isItem" type="xs:string" use="required"/>
            <xs:attribute name="originalElement" type="xs:string" use="required"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="author_Version">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="author"/>
            <xs:attribute name="begin" type="xs:date" use="required"/>
            <xs:attribute name="end" type="xs:date" use="required"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="publisher_RepItem">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="publisher_Version" minOccurs="1" maxOccurs="unbounded"/>
            <xs:attribute name="isItem" type="xs:string" use="required"/>
            <xs:attribute name="originalElement" type="xs:string" use="required"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="publisher_Version">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="publisher"/>
            <xs:attribute name="begin" type="xs:date" use="required"/>
            <xs:attribute name="end" type="xs:date" use="required"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```
<xs:complexType>
  <xs:element name="related_items_RepItem">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="related_items_Version" minOccurs="1" maxOccurs="un bounded"/>
      </xs:sequence>
      <xs:attribute name="isItem" type="xs:string" use="required"/>
      <xs:attribute name="originalElement" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:complexType>
</xs:element>

<xs:element name="related_items_Version">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="related_items"/>
    </xs:sequence>
    <xs:attribute name="begin" type="xs:date" use="required"/>
    <xs:attribute name="end" type="xs:date" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="FAX_number" type="xs:string"/>

<xs:element name="ISBN" type="xs:string"/>

<xs:element name="attributes">
  <xs:complexType>
    <xs:all>
      <xs:element ref="ISBN"/>
      <xs:element ref="number_of_pages"/>
      <xs:element ref="type_of_book"/>
      <xs:element ref="size_of_book"/>
    </xs:all>
  </xs:complexType>
</xs:element>

<xs:element name="author">
  <xs:complexType>
    <xs:all>
      <xs:element ref="name">
        <xs:complexType>
          <xs:all>
            <xs:element ref="first_name"/>
            <xs:element ref="middle_name"/>
            <xs:element ref="last_name"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
      <xs:element ref="date_of_birth"/>
      <xs:element ref="biography"/>
      <xs:element name="contact_information">
        <xs:complexType>
          <xs:all>
            <xs:element name="mailing_address">
              <xs:complexType>
                <xs:all>
                  <xs:element ref="street_information"/>
                  <xs:element ref="name_of_city"/>
                  <xs:element ref="name_of_state"/>
                  <xs:element ref="zip_code"/>
                  <xs:element name="name_of_country" type="xs:string"/>
                </xs:all>
              </xs:complexType>
            </xs:element>
            <xs:element ref="phone_number"/>
            <xs:element ref="email_address"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:all>
  </xs:complexType>
</xs:element>
</xs:element>
</xs:complexType>
</xs:element>

<xs:element name="authors">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="author_RepItem" minOccurs="1" maxOccurs="un bounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="biography" type="xs:string"/>

<xs:element name="temporalRoot"/>
<xs:element name="related_items">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="related_item" minOccurs="0" maxOccurs="5"/>
    </xs:sequence>
    <xs:attribute name="related_items_id" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="size_of_book">
  <xs:complexType>
    <xs:all>
      <xs:element ref="length"/>
      <xs:element ref="width"/>
      <xs:element ref="height"/>
    </xs:all>
  </xs:complexType>
</xs:element>

<xs:element name="street_address" type="xs:string"/>

<xs:element name="street_information">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="street_address" maxOccurs="2"/>  
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="subject" type="xs:string"/>

<xs:element name="suggested_retail_price">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:decimal">
        <xs:attribute name="currency" type="xs:string" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name="thumbnail">
  <xs:complexType>
    <xs:all>
      <xs:element ref="data"/>  
    </xs:all>
  </xs:complexType>
</xs:element>

<xs:element name="title" type="xs:string"/>

<xs:element name="type_of_book" type="xs:string"/>

<xs:element name="when_is_available" type="xs:date"/>

<xs:element name="width">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:decimal">
        <xs:attribute name="unit" type="xs:string" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name="zip_code" type="xs:string"/>

</xs:schema>
A.5  Temporal XML Schema with Added Constraints

```xml
<?xml version="1.0" encoding="utf-8"?>
<temporalSchema
    xmlns="http://www.cs.arizona.edu/tau/TSSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.cs.arizona.edu/tau/TSSchema/etc/TSSchema.xsd">
    <conventionalSchema>
        <include schemaLocation="DCSD.xsd" />
    </conventionalSchema>

    <annotationSet>
        <logical>

            <!-- Non-sequenced Identify Constraint: -->
            <!-- Item IDs are unique for books and may not ever be re-used. -->
            <item target="catalog">
                <nonSeqKey
                    name="bookIDKey" dimension="validTime" evaluationWindow="lifetime">
                    <selector xpath="item" />
                    <field xpath="@id" />
                </nonSeqKey>
            </item>

            <!-- Non-sequenced Referential Integrity: -->
            <!-- A related item should refer to a valid item (may not currently be an item in print). -->
            <item target="catalog/item">
                <nonSeqKeyref
                    name="relatedItemRI" refer="bookIDKey" dimension="validTime">
                    <selector xpath="." />
                    <field xpath="related_items/related_item/item_id" />
                </nonSeqKeyref>
            </item>

            <!-- Non-sequenced Cardinality: -->
            <!-- In any given year, an item may have up to 6 authors. -->
            <item target="catalog/item">
                <nonSeqCardinality
                    name="bookAuthorsNSeq" minOccurs="1" maxOccurs="6" dimension="validTime"
                    evaluationWindow="year" slideSize="year">
                    <selector xpath="." />
                    <field xpath="authors/author" />
                </nonSeqCardinality>
            </item>

            <item target="catalog/item/authors/author" />
            <item target="catalog/item/publisher" />
            <item target="catalog/item/related_items" />
        </logical>

        <physical>
            <default>
                <format plugin="XMLSchema" granularity="days" />
            </default>
            <!-- default timestamps are at logical items -->
        </physical>
    </annotationSet>
</temporalSchema>
```
B Relational Schemas

B.1 Non-temporal Relational Schemas

```sql
CREATE TABLE item (
  id CHARACTER(10) NOT NULL,
  ISBN CHARACTER VARYING(20),
  title CHARACTER(100),
  subject CHARACTER VARYING(200),
  number_of_pages INTEGER,
  type_of_book CHARACTER(50),
  length FLOAT,
  width FLOAT,
  height FLOAT,
  suggested_retail_price DECIMAL(10,2),
  cost DECIMAL(10,2),
  when_is_available DATE,
  quantity_in_stock INTEGER,
  date_of_release DATE,
  description CHARACTER VARYING(500))
ALTER TABLE item ADD PRIMARY KEY (id);
```

Figure 23: The item table creation and constraint definitions in SQL.

```sql
CREATE TABLE author(
  author_id CHARACTER(10) NOT NULL,
  first_name CHARACTER(50),
  middle_name CHARACTER(50),
  last_name CHARACTER(50),
  date_of_birth DATE,
  biography CHARACTER VARYING(500),
  street_address CHARACTER VARYING(100),
  name_of_city CHARACTER VARYING(50),
  name_of_state CHARACTER VARYING(50),
  zip_code CHARACTER(8),
  name_of_country CHARACTER VARYING(50),
  phone_number CHARACTER VARYING(50),
  email_address CHARACTER VARYING(100))
ALTER TABLE item_author ADD PRIMARY KEY (author_id);
```

Figure 24: The author table creation and constraint definitions in SQL.

```sql
CREATE TABLE item_author (
  item_id CHARACTER(10) NOT NULL,
  author_id CHARACTER(10) NOT NULL)
ALTER TABLE item_author ADD PRIMARY KEY (item_id, author_id);
ALTER TABLE item_author ADD FOREIGN KEY (item_id) REFERENCES item(id));
ALTER TABLE item_author ADD FOREIGN KEY (author_id) REFERENCES author(author_id));
```

Figure 25: The item_author table creation and constraint definitions in SQL.
CREATE TABLE item_publisher (  item_id CHARACTER(10) NOT NULL,  publisher_id CHARACTER(10) NOT NULL)

ALTER TABLE item_publisher ADD PRIMARY KEY (item_id, publisher_id);
ALTER TABLE item_publisher ADD FOREIGN KEY (item_id) REFERENCES item(id));
ALTER TABLE item_publisher ADD FOREIGN KEY (publisher_id) REFERENCES publisher(publisher_id));

Figure 26: The item_publisher table creation and constraint definitions in SQL.

CREATE TABLE publisher (  publisher_id CHARACTER(10) NOT NULL,  name CHARACTER VARYING(100),  street_address CHARACTER VARYING(100),  name_of_city CHARACTER VARYING(50),  name_of_state CHARACTER VARYING(50),  zip_code CHARACTER(8),  exchange_rate FLOAT,  currency CHARACTER(50),  phone_number CHARACTER VARYING(50),  web_site CHARACTER VARYING(40),  FAX_number CHARACTER VARYING(20) NOT NULL)

ALTER TABLE publisher ADD PRIMARY KEY (publisher_id);

Figure 27: The publisher table creation and constraint definitions in SQL.

CREATE TABLE related_item (  item_id CHARACTER(10) NOT NULL,  related_id CHARACTER(10) NOT NULL)

ALTER TABLE related_item ADD PRIMARY KEY (item_id, related_id);
ALTER TABLE related_item ADD FOREIGN KEY (item_id) REFERENCES item(id));
ALTER TABLE related_item ADD FOREIGN KEY (related_id) REFERENCES item(id));

Figure 28: The related_items table creation and constraint definitions in SQL.
B.2 Temporal Relational Schemas

To accommodate temporal relational data, we must add valid time columns to each relation and alter the primary keys so that they contain the `begin_time` column. Namely, the following changes must be made to the relational schemas presented in Appendix B.1.

```
ALTER TABLE item ADD VALIDTIME (DATE);
ALTER TABLE author ADD VALIDTIME (DATE);
ALTER TABLE publisher ADD VALIDTIME (DATE);
ALTER TABLE related_item ADD VALIDTIME (DATE);
ALTER TABLE item_author ADD VALIDTIME (DATE);
ALTER TABLE item_publisher ADD VALIDTIME (DATE);
```

```
ALTER TABLE item ADD PRIMARY KEY (id, begin_time)
ALTER TABLE author ADD PRIMARY KEY (author_id, begin_time)
ALTER TABLE item_author ADD PRIMARY KEY (item_id, author_id, begin_time)
ALTER TABLE publisher ADD PRIMARY KEY (publisher_id, begin_time)
ALTER TABLE item_publisher ADD PRIMARY KEY (item_id, publisher_id, begin_time)
ALTER TABLE related_item ADD PRIMARY KEY (item_id, related_id, begin_time)
```

Figure 29: The `item` table creation and constrain definitions in SQL.
C Example Parameters File for \( \tau \) Generator

INITIAL_PERCENTAGE 10
START_TIME 2010-01-01
FOREVER_TIME 2099-12-31
TIME_STEP 7
REQUIRED_CHANGES 36400
ITEM_PER_TIME_STEP_INSERTS 58
ITEM_PER_TIME_STEP_DELETES 58
ITEM_PER_TIME_STEP_UPDATES 58
AUTHOR_PER_TIME_STEP_INSERTS 58
AUTHOR_PER_TIME_STEP_DELETES 58
AUTHOR_PER_TIME_STEP_UPDATES 58
PUBLISHER_PER_TIME_STEP_INSERTS 58
PUBLISHER_PER_TIME_STEP_DELETES 58
PUBLISHER_PER_TIME_STEP_UPDATES 58
RELATED_PER_TIME_STEP_INSERTS 59
RELATED_PER_TIME_STEP_DELETES 59
RELATED_PER_TIME_STEP_UPDATES 59
CHANGE_TYPE gaussian
CHANGE_STDDEV 10
OUTPUT_DIR output
INPUT_FILE catalog.xml
NULL_VALUE ""
D  DB2 differences

DB2 [29] differs from the PSM standard in the following ways.

- **Temporary tables.** Instead of creating a temporary table and populating it from an existing table, DB2 requires two steps:

  ```sql
  DECLARE GLOBAL TEMPORARY TABLE table_name
  (SELECT * FROM original_table) DEFINITION ONLY
  INSERT INTO SESSION.table_name
  SELECT * FROM original_table
  ```

- **DECLARE statement.** In a DB2 user defined function (UDF), DECLARE is limited to declarations of variables and cursors only.

- **Function returning table type.** In a UDF definition, in order to return a table, the keyword RETURNS TABLE must be used.

- **Using a function that returns table in FROM clause.** In order to perform the following,

  ```sql
  SELECT ... FROM table1 t1, foo() ...
  ```

  the following syntax must be used.

  ```sql
  SELECT ... FROM table1 t1, TABLE(foo()) t2 ...
  ```

- **Input type casting.** To invoke a function or a procedure, input arguments needs to be explicitly cast. For instance, to invoke `foo('I2', '1990-07-12')` in DB2, one must specify the following.

  ```sql
  foo(CAST('I2' AS CHAR(10)), CAST('1990-07-12' AS DATE))
  ```

- **ATOMIC keyword.** When defining a function body, the BEGIN statement should be associated with the ATOMIC keyword. This is to make sure the actions performed in the function can be completely rolled-back if the calling transaction aborts.

- **Returning booleans.** Returning boolean types from a function and returning boolean arguments as OUT parameters in procedure are not supported. Alternatively, integers can be used to simulate boolean values.

- **Printing messages.** There is no PRINT statement in DB2. To print ad-hoc messages, one must use the following.

  ```sql
  SIGNAL SQLSTATE '80000' SET MESSAGE_TEXT = 'message';
  ```

  Moreover, if the message contains multiple pieces that are concatenated, e.g., several variables, the following needs to be specified. Note that non-string typed values need to be cast. "||" is the concatenation operator.

  ```sql
  DECLARE output_text CHARACTER(250);
  SET output_text = var1 || ' ' ||
  CAST(time1 AS CHARACTER(20)) || ' ' ||
  CAST(time2 AS CHARACTER(20)));
  SIGNAL SQLSTATE '80000' SET MESSAGE_TEXT = output_text;
  ```