SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **00**:1–29
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Benchmark Frameworks and τBench

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

SUMMARY

Software engineering *frameworks* tame the complexity of large collections of classes by identifying structural invariants, regularizing interfaces, and increasing sharing across the collection. We wish to appropriate these benefits for families of closely related benchmarks, say for evaluating query engine implementation strategies. We introduce the notion of a *benchmark framework*, an ecosystem of benchmarks that are related in semantically-rich ways and enabled by organizing principles. A benchmark framework is realized by iteratively changing one individual benchmark into another, say by modifying the data format, adding schema constraints, or instantiating a different workload. Paramount to our notion of benchmark frameworks are the ease of describing the differences between individual benchmarks and the utility of methods to validate the correctness of each benchmark component by exploiting the overarching ecosystem. As a detailed case study, we introduce τBench, a benchmark framework consisting of ten individual benchmarks, spanning XML, XQuery, XML Schema, and PSM, along with temporal extensions to each. A second case study examines the *Mining Unstructured Data* benchmark framework and a third examines the potential benefits of rendering the TPC family as a benchmark framework. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Benchmarks, XML, temporal databases

1. INTRODUCTION AND MOTIVATION

Benchmarks enable an apples-to-apples comparison of competing software or hardware systems, architectures, algorithms, or other technologies, as well as an evaluation of a given technology under different scenarios. Benchmarks generally consist of *data*, a *schema* to describe the data, and *workload operations* to perform on the data. (Benchmarks vary on the emphasis each accords these three components.) A benchmark provides a *standard* with which to compare and evaluate. For example, the Standard Performance Evaluation Corporation (SPEC) has created a set of commercial benchmarks for testing CPU performance, with the main goal of providing compiler vendors with representative programs [29]. (As SPEC elegantly asserts, "an ounce of honest data is worth a pound of marketing hype." [58])

Typical uses of a benchmarks include [27]:

- Comparing different software and hardware systems.
- Comparing software products on one machine.
- Comparing machines in a compatible family.
- Comparing releases of a product on one machine.
- Comparing implementation strategies of a system.

*Correspondence to: S. W. Thomas, School of Computing, Queen's University, Kingston, ON, Canada K7L 3N6.

It is a challenging task to create datasets, schemas, and workloads that are both simple enough to be understood by a wide audience as well as realistic enough to be useful. Additionally, since major technical and marketing decisions often hinge on the results of a benchmark evaluation, it is critical that the benchmarks be correct. Any method that helps in the creation, maintenance, and description of benchmarks will thus be beneficial to all stake holders involved.

In the software engineering domain, *frameworks* are widely used to manage the complexity of creating and maintaining software functionality [19]. Each framework includes a set of organizing principles that enable it to be assimilated and used effectively [32].

In this paper, we introduce the notion of a *benchmark framework*, which emphasizes the organizational principles of frameworks to achieve many of the advantages that software engineering frameworks provide. Briefly, a benchmark framework is an ecosystem of benchmarks that are related in meaningful ways, along with shared tools and techniques to generate and validate new benchmarks from existing benchmarks. Benchmark framework can help alleviate the difficulties of creating, maintaining, validating, and describing sets of related benchmarks. Benchmark frameworks are built iteratively by using tools to alter existing benchmarks to obtain new benchmarks, followed by a validation phase to ensure that each new benchmark is still correct. This iteration produces a DAG-like structure of benchmarks, where nodes are benchmarks and edges are the relationships between benchmarks. The structure of a benchmark framework encourages the designer to be explicit about the specific relationships between the individual benchmarks, thereby highlighting the commonalities between and distinctions among the benchmarks.

Not every benchmark needs to be in a framework, and benchmark frameworks may not be ideal for every situation. For example, if commercial politics dictate exactly which benchmarks must be used to evaluation a given technology, and how those benchmarks must be structured, then benchmark frameworks may not be appropriate. However, benchmark frameworks have several advantages of which practitioners and researchers can take advantage, such as (i) reusing an extant benchmark without requiring modification of that benchmark, which helps to reduce development cost, (ii) sharing of data, schema, workloads, and tools across benchmarks, (iii) regularizing the execution of the benchmarks, and (iv) clearly communicating the essence of each benchmark. These advantages will encourage practitioners and researchers to create their own benchmarks, based on existing benchmarks, when they may have otherwise avoided doing so due to time and resource constraints. In this paper, we explore these potential advantages of benchmark frameworks.

Consider the TPC benchmarks, a popular family of OLTP and OLAP benchmarks [69]. In their current form, the TPC benchmarks are loosely connected, but understanding these connections is difficult. For example, documentation suggests that the TPC-R benchmark is related to the TPC-H benchmark, but exactly how is not defined. (Do the two use the exact same data? Same schema? Same workload? Do the data generators share code?) Further, the TPC-E benchmark seems to be related, in different ways, to both the TPC-C and TPC-H benchmarks. Again, since the relationship is not explicitly defined, potential users of these benchmarks can only guess. As we show in Section 7, additional structuring, documentation, and sharing of tools can realize a TPC benchmark framework that embodies the advantages listed above.

To make our proposal concrete, we present τBench, a benchmark framework for temporal databases comprised of some ten individual benchmarks. τBench emphasizes the organizational principles of using statement modifiers [8], commutativity validation, and temporal upward compatibility [5] for creating temporal benchmarks from non-temporal ones.

The contributions of this paper are as follows:

- We introduce and describe in detail the notion of a *benchmark framework*, which allows simple, correct, and adaptable sets of related benchmarks (in any domain) to be created and documented.
- We show the utility and generality of benchmark frameworks.
- We present τBench, a benchmark framework which encompasses several XML, relational, and temporal technologies. τBench is distributed according to the GNU Lesser GPL [22] and is available online [64].

DOI: 10.1002/spe

- We briefly present the MUD and TPC benchmark frameworks from differing domains, to emphasize the generality and utility of our approach.

The remainder of this paper is structured as follows. We describe software engineering frameworks, our inspiration for benchmark frameworks, in Section 2. We describe benchmarks and benchmark families, as well as introduce benchmark frameworks, in Section 3. We introduce benchmark frameworks in Section 4. We present τBench, a specific instance of a benchmark framework, in Section 5. We briefly present two additional benchmark frameworks from other domains in Sections 6 and 7. Based on our experience from the case studies, we discuss the advantages and disadvantages of benchmark frameworks in Section 8. Finally, we conclude and outline future work in Section 9.

2. SOFTWARE ENGINEERING FRAMEWORKS

As mentioned previously, developers use software engineering frameworks to manage the complexity of creating and maintaining software functionality [19, 32]. For example, the Java Swing framework provides Java developers with a standardized API for creating graphical user interfaces [74]. This standardization allows developers to easily develop new GUIs and to easily understand existing GUIs written with Swing. Other frameworks include: the C++ Standard Template Library (STL) [4], an assemblage of data structures; OpenMS [61], for developing mass spectrometry data analysis tools; the Computational Crystallography Toolbox [28], for helping to advance the automation of macromolecular structure determination; ControlShell [52], for developing real-time software; and log4j [3], for enabling logging with context to Java programs through a hierarchy of loggers.

Software engineering frameworks are typically born from the necessity of taming the complexity that grows naturally over time. Developers begin by writing single classes. Families of related classes emerge as the application grows, such as those for sorting and searching a custom data structure. Over time, the family of related classes grows so large and interconnected that they become unmanageable and unmaintainable without some explicit structure and organization. At this point, developers create frameworks to tame this complexity: structural invariants are identified, interfaces are regularized, constraints are identified and enforced, and the coherency of the family is emphasized.

The creation of frameworks is not mechanical: it involves the manual and creative analysis of the problem domain and desired solution space. The result, however, is often a significant decrease in complexity of the classes and a significant increase in their reusability. In addition, holes in functionality can be easily identified and filled.

Let's examine in some detail the Java Collections framework [43], a large assemblage of data structures. The Collections framework is comprised of a sequence of successively elaborated Java interfaces to characterize functionality, a sequence of successively elaborated Java abstract classes that implement these interfaces to provide data structure-independent methods, and a set of concrete classes that extend these abstract classes to provide specific data structures and thus trade-offs in performance between the various operations specified originally in the interfaces. The Collections framework also uses new concepts such as modifiable versus immutable, ordered versus unordered, getters versus mutators, comparables and compareTo, and static classes (e.g., `Arrays`). The framework exploits Java's inheritance, generics, enhanced `for` loops, and dynamic binding language features. Finally, the framework effectively utilizes the Command, Iterator, and Template patterns [23].

Within this framework, the `Iterable` Java interface provides a method to obtain an `Iterator` across the collection (hence, every collection has an iterator). The `Collection` Java interface extends `Iterator` and introduces a dozen methods (functionality) used across the framework. The `List` Java interface extends the `Collection` Java interface and specifies the functionality of an ordered collection through ten additional methods, including the `set(int <E>)` method that explicitly states a position. The `AbstractList` class provides initial implementations of most

DOI: 10.1002/spe

of the methods of the `List` interface, often utilizing the Template Pattern. Finally, we get to the `LinkedList` concrete class (which can actually be instantiated); this class provides a specific data structure and refined implementations by overriding some of the implementations in the `AbstractList` class, thereby providing specific performance properties, such as a very fast *next* accessor and minimal storage overhead for highly volatile collections.

The framework thereby presents a unified and carefully structured assemblage of some 75 classes and interfaces supporting several thousand methods, enabling programmers to understand and use these many classes without having to study the details of every individual Java method. As another benefit, this carefully orchestrated dance of Java interfaces, abstract classes, and concrete classes helped to ferret out inconsistencies. For instance, the documentation for the `Vector` class states, "As of the Java 2 platform v1.2, this class was retrofitted to implement the List interface, making it a member of the Java Collections Framework." The `Collection` and `List` interfaces make it easier for programmers to use these APIs, as all the data structures in this framework abide by the `Collection` interface and all the ordered data structures abide by the `List` interface.

In this paper, we apply what has been performed repeatedly in software engineering to the domain of benchmarks. Our goal is to achieve the structure and organization that software engineering frameworks provide, except using individual benchmarks and families of related benchmarks instead of individual classes and families of related classes.

3. BENCHMARKS AND BENCHMARK FAMILIES

We now define more thoroughly the concepts and structures of individual benchmarks and benchmark families, on our way to the more internally-structured benchmark frameworks in the following section.

3.1. Benchmarks

A *benchmark* is a collection of data, workload, and metrics. Individual benchmarks, such as CloudCMP [34], INEX [31], WSTest [13], X007 [35], XMark [50], and XQBench [20], allow applications and algorithms to be compared and evaluated in a standard, consistent setting, giving researchers and practitioners invaluable guidance as to the relative performance of the systems under consideration [27].

In this paper, we further structure an individual benchmark into ten components, shown in Table I. All but the last component, tools, are required, though some might be only implied by a particular benchmark. (We recommend though that a benchmark be explicit about every component.) In the table, a component is ordered after its predecessor(s), so for example the data model must be specified before the data definition language, which must in turn be specified before the schema, which must be specified before the actual data of the benchmark.

The *schema* component describes the structure and constraints of the data, such as the relationship between books and authors (from the aforementioned XBench example). The schema is expressed in a *data definition language (DDL)*, also a component, such as XML Schema, SQL DDL, a new schema language, or informally. The schema component must conform to the benchmark's specified DDL.

The *data* component is the actual data in the benchmark, for example an XML file containing information for a library catalog of books and authors. The data can be an actual dataset, or a tool that randomly generates data, or a combination of the two. For example, in XBench, XMark, and XOO7, the data is randomly generated using ToXgene, xmlgen, or in-house tools, respectively; in TPC-H (part of the TPC benchmark family, to be discussed in Section 3.2), there is a given single (static) dataset. The data conforms to a *data model*, which describes the structure and form of the data. For example, the data in XMark is represented in XML thus conforming to the hierarchical data model. Within a benchmark, the data must conform to the data model.

The *workload* component contains tasks to perform on the data, such as querying for specific authors, inserting new books, or ensuring that all books are associated with at at least one author.

DOI: 10.1002/spe

Table I. The ten components of a benchmark.

Component Name	Description	Example
1. Data model	Describes how the data is stored and organized.	XML model
2. Data definition language (DDL)	The language in which the schema is written.	XML Schema
3. Schema	Describes the structure of and constraints on the data.	A book has at most 5 authors
4. Data	The actual data of the benchmark.	A catalog with books and authors
5. Data manipulation language (DML)	The language in which the workload is written.	XQuery
6. Workload	A set of tasks (e.g., queries) to perform on the data.	A query which selects books by "Ben"
7. Metrics	The metrics to be measured and how to measure them.	Milliseconds per query
8 Context	A specification of the execution environment.	CPU, memory, and OS version
9. Documentation	Metadata (authors, date, version) and execution guide.	Author: Bob Smith.
10. Tools	Executables.	Makefiles; data generators and validators; execution scripts; etc.

The workload is expressed in a *data manipulation language (DML)*, also a component, such as XQuery, SQL DML, or a new query language.

The *metrics* component is a description of the metrics to be measured, including instructions on how to measure and calculate each one. For example, DC/SD XBench contains two metrics: bulk load time and query execution time.

The *context* component specifies what characteristics of the software and hardware environments must be reported during the execution of the benchmark. The benchmark creator can be as specific as desired: the context could require that the exact CPU characteristics (e.g., clock speed and cache size) must be reported, or simply the number of cores.

The *documentation* component includes metadata and an execution guide. Metadata includes the benchmark name, authors and affiliations, date published and date updated, version, and other descriptive information. The execution guide describes how to configure the software environment, which settings and values are required, order of program execution, etc.

The metrics, context, and documentation form the *provenance* of both the benchmark and the results of applying that benchmark to a particular hardware/software configuration. We anticipate that benchmarks will evolve through distinct and identified versions, with a benchmark result explicitly associated with a version.

Various kinds of benchmarks will emphasize different components. Database benchmarks emphasize the data and workload and often deemphasize the metrics (e.g., simply transactions per second). Compiler benchmarks like SPEC have programs as their workload, with little emphasis on the data and even less on the DDL and schema. It appears that all the benchmarks mentioned in this paper contain (or could have contained, if the benchmark was more fully realized) all of the above-mentioned components.

Finally, the *tools* are (optional) executables for generating and validating the data, translating and executing the workload, and collecting metrics. Example tools include the following.

Generation. Data generators, such as ToXgene [6] and τGenerator (Section 5.6.1), turn raw input data into larger and more complex datasets by repeating, combining, and changing the raw data in various ways, perhaps based on user parameters. Data generators are often template based, generating data from a user-specified template.

Validation. Validation scripts ensure that component changes are correct, amongst other things (Section 4.4).

DOI: 10.1002/spe

Translation. Data translators are used to easily change data from one format to another, for example by shredding an XML document into relations [53]. Schema/workload translators such as TXL [12] can be used to translate the schema and workload from one form to another, for example to translate τXQuery [26] queries into standard XQuery [73] queries [56].

3.2. Benchmark Families

A *benchmark family* is a set of loosely related benchmarks. A benchmark family is often developed by a single organization. Benchmark families lack the information necessary to easily identify benchmark relationships or tools necessary to derive new benchmark components from existing benchmark components.

An example of a benchmark family is XBench [76], which contains four benchmarks: Document-centric (DC)/Single Document (SD), DC/Multiple-Document(MD), Text-centric (TC)/SD, and TC/MD. The benchmarks are loosely related in that they share a document generation tool. However, the benchmarks do not share schemas or workloads, and the specific relationships between benchmark components is not well defined.

Another benchmark family is TPC, which we described in Section 1. As in XBench, the benchmarks in TPC are loosely related in that they are developed by the same organization, but the relationships between benchmark components is not well defined and no data, schema, or workloads are re-used among the currently supported TPC-C, TPC-H, and TPC-E benchmarks.

SPEC provides a wide rage of evolving benchmarks [29, 57]. For example, the programming language community employs the *SPEC CPU benchmark* to study program performance with intensive computation. This benchmark stresses not only the processor of a machine, but also its memory hierarchy and the properties of the compiler. Other SPEC benchmarks include SPECviewperf to evaluate graphics performance, SPECapc to evaluate performance of specific applications, such as LightWave 3D and SolidWorks, SPEC MPI to evaluate MPI-parallel, floating point, compute-intensive across cluster and SMP hardware performance, SPEC OMP to evaluate OpenMP-parallel shared-memory parallel processing across shared-memory parallel processing, SPECjvm to evaluate performance of the Java Runtime Environment, SPECjbb as well as SPECjEnterprise and SPECjms to evaluate performance of Java business applications, and SPECsfs to evaluate performance of file servers.

The TREC conference series [70] has produced a set of text retrieval benchmarks, each consisting of a set of documents (data), a set of topics (questions), and a corresponding set of relevant judgments (right answers), across a range of domains, such as blog, chemical, genomics, legal, and web, and across question characteristics, such as filtering, interactive, novelty, and query.

4. BENCHMARK FRAMEWORKS

We introduce here a *benchmark framework* as a refined benchmark family. A benchmark framework is an ecosystem of benchmarks that are inter-connected in semantically-rich ways. For example, the data used in one benchmark might be derived from the data of another (e.g., by increasing the size of the data), while the workloads and schemas of the two benchmarks are identical.

Like software frameworks, the creation of a benchmark framework is not mechanical. Instead, they are created iteratively: beginning with a root (or *foundation*) benchmark B, a new benchmark B' is created by changing one or more of the six components of B. (Components are listed in Table I and described in Section 3.1.) A *relationship* between B and B' is made to describe the changes. In this way, a DAG of benchmarks emerges, where nodes are benchmarks and each edge is the relationship between each pair of benchmarks. This graph exactly describes a *benchmark framework*. At this point, benchmark frameworks depart in design from software frameworks such as the Java Collections framework. To build the Collections framework, the designers had to first boil the ocean: almost the entire framework had to be in place before any of it was useful. In contrast, benchmark frameworks are useful immediately, as new benchmarks can easily be derived whether there are two benchmarks or two hundred present in the framework.

DOI: 10.1002/spe

Figure 1 shows an example benchmark framework, which we use as a running example in this section. The framework begins with the DC/SD XBench benchmark, a popular XQuery benchmark from the XBench family of benchmarks [76]. The data in the DC/SD XBench benchmark is a library catalog of books and authors contained in a single XML document. In this simple benchmark framework, the τXBench benchmark is derived from the DC/SD XBench benchmark by using a tool to render the data as time-varying, while the PSM benchmark, which is utilized in a study to evaluate the performance of the SQL/PSM language constructs, is derived from the DC/SD XBench benchmark by using a tool to shred the XML data into relational data. The arcs and annotations in the DAG succinctly denote these relationships; we describe these annotations in detail in Section 4.2.

We now describe the general principles of benchmark frameworks (Section 4.1), define the possible relationships between benchmark components and how to visualize relationships (Section 4.2), define relationship semantics (Section 4.3), and present the methodology for creating a benchmark framework (Section 4.4).

4.1. General Principles

Our proposal of benchmark frameworks, detailed in the following sections, is based on several general principles. We define a benchmark framework to be a collection of benchmarks that satisfy all five principles.

First a benchmark framework must explicitly state one or more ***organizing insights*** that bring together constituent benchmarks into a coherent whole. For example, the Java Collection Framework discussed in Section 2 utilizes several such insights: successively elaborated and related Java interfaces, abstract classes, and concrete classes, utilizing several powerful design patterns throughout, and effectively exploiting particular Java constructs. As we describe three benchmark frameworks as case studies later in this paper, we first state the organizing insight(s) for each.

The second organizing principle is that of ***automation***, of producing a component of a child benchmark by executing mapping or conversion programs on a component of a parent benchmark. While some of the components of a benchmark must be defined manually, many of the other components can be generated by a tool that automatically maps a prior component to the new component. We'll see many examples of such mapping tools.

Reuse is the third required organizing principle of a benchmark framework. Reusing existing components to create new components is one of the fundamental ideas behind frameworks of any kind, and benchmark frameworks are no different. By reusing benchmark components, developers can save development cost, benchmark relationships become more clear, and cross validation between components becomes more applicable. Reuse must be applied to several components of each constituent benchmark in the framework. We'll see many examples as we examine τBench in detail shortly, as so we'll mention just a few applications of reuse here.

The queries comprising a workload can be mapped to a new language with a source-to-source translation, either manually or with a tool. A separate source-to-source translation can map a standard language to a DBMS-specific variant. These translations can be composed, to define further benchmark components.

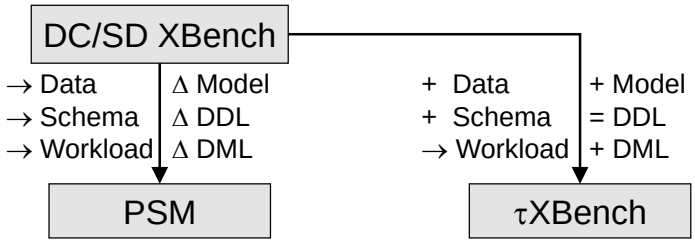

Figure 1. Example benchmark framework.

DOI: 10.1002/spe

Similarly, the mapping of the data component between two benchmarks can often be reused elsewhere in the framework to produce yet other components, when two source benchmarks are also related in a different way than to their child benchmarks. An example is the mapping of the data from XBench to PSM being reused in the mapping from τXBench to τPSM. The source-to-source mapping of a schema component can likewise be reused. A third potential for reuse is a tool. For example, τBench employs a single `queryRunner` Java application to time queries in three of its constituent benchmarks (those employing the relational model).

Organizational clarity is a fourth general principle. By clearly defining the relationships between benchmark components, developers and users of the benchmarks can immediately determine the differences and similarities between them. Choosing which benchmark or set of benchmarks to use, or which benchmark to extend, becomes much easier. These relations should be formalized in a representation exemplified in Figure 1.

Providing ***consistency and validation checking*** is the final general principle required of benchmark frameworks. As mentioned previously, users of benchmarks often have a high stake in the correctness of benchmark components, as they are used to evaluate new technologies and make critical business decisions. Benchmark frameworks allow a new level of consistency and validation checks to be performed across benchmarks, since they share components and tools. Thus, the entire benchmark framework is strengthened with each additional check of each additional component.

These five general principles work in concert to allow benchmark frameworks to be developed faster and to attain higher quality than a more loosely structured family containing the same constituent benchmarks.

4.2. Relationships Between Benchmark Components

A *relationship* between a component of a parent benchmark component and a component of a child benchmark is an explicit link between the two describing how to change one component into the other. Component relationships characterize the *flow of information* between the benchmarks. Relationships allow new benchmark components to be easily adopted from existing benchmark components and for that adoption to be explicitly and clearly defined. Each component relationship, annotating a line from one benchmark to another, is prefixed by one of the following symbols.

Expand (+). A component is expanded to include additional content. For example, the data in τXBench is expanded from DC/SD XBench by making the data temporal (i.e., adding timestamps to the data, and changing random books and authors at various points in time).

Subset ($\supset$). A component is subsetted to remove content. As we'll see in Section 5.4, the data definition language of τPSM, that of a temporal extension of the SQL DML, is subsetted back to that of SQL DML in the subsequent τPSM-MaxMapped DML.

Retain (=). A component is retained without modification. For example, the DDL remains the same (i.e., XML Schema) between the DC/SD XBench and τXBench benchmarks.

Replace (Δ). A component is replaced (i.e., completely removed) and substituted with a new instance. For example, the data model for PSM, which is the relational data model, replaces DC/SD XBench's hierarchical data model.

Map ($\rightarrow$). A component is mapped from one data model/DDL/DML to another, but the ideas behind the components remain the same. For example, the workload for PSM is logically unchanged from the DC/SD XBench benchmark (i.e., the queries are seeking the same information), but must be mapped from the XQuery syntax to the PSM DML syntax. Here the *form* of the component changes, but the *content* remains.

We visualize the relationships of a benchmark framework by displaying a DAG, where nodes are benchmarks and edges are relationships between them (e.g., Figure 1). We term this a *Benchmark-Relationship* (or *B-R*) *diagram*.

A B-R diagram enumerates the benchmarks within the framework (or family) as well as between frameworks and families. Thus the XBench family is included in Figure 1 because one of its

DOI: 10.1002/spe

benchmarks is used and extended in the τBench framework. An edge between two benchmarks denotes that components are shared between them; the edge is annotated with the exact relationship of each component. (We only annotate the first six components, since these best characterize the changes from benchmark to benchmark.) For consistency, component annotations are always displayed in the same location. For example, the data component is shown in the upper left of the edge; the DML component is shown in the bottom right. Note the left-right correspondence: the Data accords with a Data Model, the Schema is expressed in a DDL, and the Workload is expressed in a DML. For brevity, component annotations are omitted from the display if the relationship is retain (=), except in Figure 1, in which we aimed for completeness.

4.3. Relationship Semantics

A new benchmark can be comprised of components from multiple existing benchmarks (for example, the data and schema from one and the workload from another), which is precisely why a benchmark framework forms a DAG instead of a tree. In these cases, a component annotation existing from both parents implies that the components should be merged (by performing a union) into the child.

Each component can be associated with only one relationship between two benchmarks. If two relationships are needed, such as expanding and mapping a workload, this is better expressed as two steps, yielding a second benchmark derived from the original with one of the relationships, say mapped, and a third benchmark, derived from the second with the other of the relationships, in this case expanding.

When the data is expanded or subsetted, the semantics are clear: data was added or removed, respectively. But when data models are expanded or subsetted, the semantics are domain-dependent. In the temporal database domain, for example, a data model can be expanded from non-temporal relational to temporal relational. While this can also be expressed as a replacement (Δ) of data models, we feel that expanding (+) more clearly describes the relationship since the temporal data model offers similar but more information.

In general, the map relationship ($\rightarrow$) is only applicable to the data, schema, or workload components; it is not appropriate to say that a data model, DDL, or DML has been mapped, since these components contain ideas rather than content.

Finally, a change in one component may imply a change in other component(s). For example, when changing the data model from the XML hierarchical data model (where the DDL is likely to be XML Schema or Relax NG [71]) to the relational set-based data model, the DDL also must be changed to an appropriate language (e.g., SQL DDL). Table II lists these and other structural implications which help define the validity of a benchmark framework. This list is not exhaustive, but rather illustrative of the semantics of component relationships. As mentioned in Section 9 (Future Work), we envision an application-specific tool that can automatically validate component relationships, given a list of constrains such as those in Table II. For example, if the relationship between the data component of two benchmarks is subset ($\supset$), then the tool can read the datasets and ensure that one is in fact a subset of the other; similar checks can be performed for schemas and workloads.

4.4. Methodology

We now describe how to create a benchmark framework, which consists of three steps: (i) begin with a foundation benchmark; (ii) transform the foundation benchmark into a new benchmark to suite your application's needs; and (iii) validate the correctness of the data, schema, and associated constraints. Repeat the process for each additional benchmark that is required. We describe each step in turn.

Foundation. Start with an existing benchmark as a solid foundation. In our example, we use DC/SD XBench as the foundation benchmark, since it provides well-defined schema and datasets, which are based on generally realistic assumptions [76].

DOI: 10.1002/spe

Table II. Examples of relationship constraints.

Relationship(s)	Implication	Explanation
$=$ Data $\vee \supset$ Data	$=$ Data Model	If the data remains the same, or is subset, then it must have the same data model, because data is an instance of the model.
$\supset$ Data	$=$ DDL $\wedge$ ($=$ Schema $\vee \supset$ Schema)	Subsetting the data implies constraints on the schema
Δ Data Model	Δ DDL $\wedge$ Δ DML $\wedge$ ($\rightarrow$ Data $\vee \Delta$ Data)	New data models need new DDLs and DMLs, and the data must be mapped or changed to the new model.
$\rightarrow$ Data Model	($\rightarrow$ DDL $\vee \Delta$DDL) $\wedge$ ($\rightarrow$ DML $\vee \Delta$DML)	Mapping the data model implies fairly whole-scale changes to the DDL and DML.
$=$ Schema $\vee \supset$ Schema	$=$ DDL	The schema is an instance of the DDL.
$=$ DDL	$=$ Schema $\vee \supset$ Schema	–
Δ DDL	$\rightarrow$ Schema $\vee \Delta$ Schema	–
$=$ Workload $\vee \supset$ Workload	$=$ DML	The workload is an instance of the DML.
$\supset$ DML	$\rightarrow$ Workload $\vee \supset$ Workload	Subsetting the DML necessitates changes to the workload that utilizes the removed constructs.
Δ DML	$\rightarrow$ Workload $\vee \Delta$ Workload	–

Table III. A description of the τBench benchmark framework.

Benchmark Name	Derived From	Data Model	DDL	DML
DC/SD XBench	–	XML	XML Schema	XQuery
τXBench	DC/SD XBench	XML	τXSchema	τXQuery
τXSchema	τXBench	XML	τXSchema	–
PSM	DC/SD XBench	Relational	PSM DDL	PSM DML
PSM-DB2	PSM	Relational	DB2 DDL	DB2 DML
τPSM	PSM, τXBench	Relational	PSM DDL	τPSM
τPSM-MaxMapped	τPSM	Relational	PSM DDL	PSM DML
τPSM-MaxMapped-DB2	τPSM-MaxMapped	Relational	DB2 DDL	DB2 DML
τPSM-PerStmtMapped	τPSM	Relational	PSM DDL	PSM DML
τPSM-PerStmtMapped-DB2	τPSM-PerStmtMapped	Relational	DB2 DDL	DB2 DML

Transformation. Build new components from existing components. Either manually or by using automated tools, alter the components of existing benchmarks to create new benchmark components. For example, use the τGENERATOR tool (to be described in Section 5.6.1) to generate temporal data from the non-temporal data of DC/SD XBench. We term the resulting new benchmark the *child* and the existing benchmark the *parent*, respectively.

Validation. Perform cross-validation of benchmark components throughout the process (i.e., after each change). There are no standard set of validation steps applicable to all benchmark frameworks, as these steps are highly domain and task dependent. Validation steps can be simple (e.g., checking referential integrity of two relations using Perl hash tables) or complex (e.g., commutativity validation of multiple benchmarks, described in Section 5.6.3); we provide examples in each of the benchmark frameworks presented in this paper.

By performing continuous validation steps, each component is tested in different ways. Each successful validation adds validity to the entire benchmark framework, due to the rich interconnectedness of the benchmarks: when the data is validated in a child benchmark, the data is also likely to be valid in the parent. These validation steps check more than just the data: they check the schema, the data generation tools, the data and query translation tools, and the query engine. The resulting ecosystem of benchmarks is now stronger than the sum of the individual benchmarks, due to the validation of each component under multiple perspectives.

DOI: 10.1002/spe

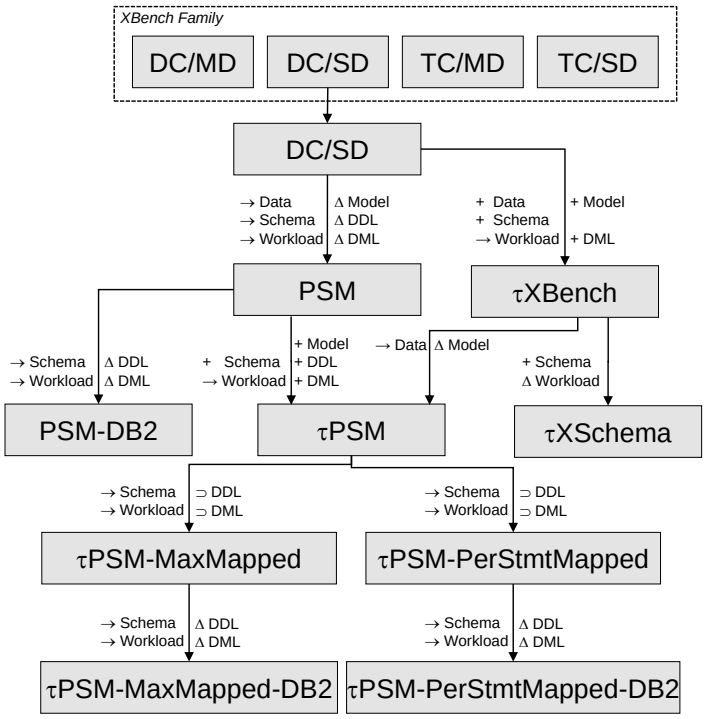

Figure 2. The Benchmark-Relationship diagram of the τBench framework.

5. CASE STUDY: τBENCH

Using the methodology just presented, we have created a benchmark framework, called τBench [64, 68]. As we describe below, τBench makes use of the organizational insight of using statement modifiers, commutativity validation, and temporal upward compatibility when deriving temporal benchmarks from non-temporal benchmarks.

In this section, we first describe the structure of τBench. We then outline the tools we have built to support the creation and validation of the individual benchmarks of τBench. Finally, we provide a detailed example of the creation of one of the benchmarks in the framework.

τBench currently consists of ten individual benchmarks, spanning various technologies including XML, XML Schema [40, 41], XQuery [73], τXSchema [14–16, 33, 55, 63], τXQuery [24–26], PSM [18,39], and τPSM [24,56]; the Benchmark-Relationship diagram is given in Figure 2 and the main components are summarized in Table III. The benchmarks are fully described elsewhere [68]; we briefly describe the components of each here.

5.1. *The* DC/SD XBench *Benchmark*

XBench is a family of benchmarks for XML DBMSes, as well as a data generation tool for creating the benchmarks [76]. The XBench family has been widely used as standard datasets to investigate the functionalities of new and existing XML tools and DBMSes. (Other XML benchmarks include MBench [49], MemBeR [1], Transaction Processing over XML (TPoX) [44], XMach-1 [9], XMark [51], XPathMark [21], and XOO7 [11]. Most of these benchmarks focus on XQuery processing, as does XBench.)

To create the 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 independent dimensions: application characteristics (i.e., whether

DOI: 10.1002/spe

```
<catalog>
  <item id="I2">
    ...
    <authors>
      <author id="A5"> ... </author>
      ...
    </authors>
    <publisher id="P6"> ... </publisher>
    <related_items>
      <related_item id="R1"> ... </related_item>
    </related_items>
  </item>
  ...
</catalog>
```

Figure 3. An outline of the XBench DC/SD schema.

the database is data-centric or text-centric) and data characteristics (i.e., single-document vs. multi-document). Thus, XBench consists of four different categories that cover DC/SD, DC/MD, TC/SD, and TC/MD, respectively.

The DC/SD XBench benchmark is one of the four constituent benchmarks of the XBench family, based strictly on the DC/SD category. We use this benchmark as the foundation of τBench, due to its combination of simplicity and generally realistic assumptions.

Schema. The schema of DC/SD XBench is specified as an XML Schema [40, 41]. The schema defines the library catalog, where `<item>` (i.e., book) elements contain a list of `<author>`s, a `<publisher>`, and a list of `<related_items>` of related books (Figure 3). Each `<item>`, `<author>`, and `<publisher>` has a unique `id` attribute. The schema specifies several constraints, such as the uniqueness of the `id` attributes and that items can be related to between 0 and 5 other items.

Data. XBench populates each data element and attribute with random data generated from ToXgene [6]. ToXgene is a template-based tool that generates synthetic XML documents according to one or more user-defined templates.

The data in DC/SD XBench consists of a single document, `catalog.xml`, which contains information about a library catalog, i.e., items (books), authors, publishers, and related items. The actual text content of each 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>` of an `<author>` could be the string "BABABABAOGREAT". (However, some content is not random. For example, a `<related_ID>` element will always point to an existing item ID.)

A crucial point to note is that, because of the data generation process, there is a strict one-to-many relationship between items and authors: 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 items and publishers: an item has exactly one publisher, and each publisher publishes exactly one item. These constraints are not, however, explicit in the schema, but must be satisfied by tools that manipulate the data, as we shall see in Section 5.6.1.

Workload. The workload consists of 17 XQuery [73] queries. Each query tests a different feature of the query engine, such as existential quantifiers (query *q6*: "Return items where some authors are from Canada"), sorting by string types (query *q10*: "List the item titles ordered alphabetically by publisher name"), and uni-gram searching (query *q17*: "Return the ids of items whose descriptions contain the word hockey"). Each such query can thus be considered a *microbenchmark* [49], designed to test the performance of a specific language construct. Other microbenchmarks include IMbench [38] (addressing system latency), MemBeR [1] (also XQuery), and MBench [49] (also XQuery).

DOI: 10.1002/spe

5.2. *The* τXBench *Benchmark*

This benchmark adds a temporal dimension to the XBench DC/SD benchmark, so that temporal databases and query languages can be evaluated.

In particular, τXQuery is a temporal query language that adds temporal support to XQuery by extending its syntax and semantics [24–26]. τXQuery moves the complexity of handling time from the user/application code to the query processor. τXQuery adds two temporal statement modifiers [8] 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, τXQuery 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.

The τXBench benchmark was created to investigate τXQuery and other temporal query languages.

Schema. The schema of τXBench is reused from XBench DC/SD and translated into a *temporal XML schema* [14, 15], which is the same as an XML Schema, except with annotations to specify which elements (i.e., books, authors, and publishers) may vary over time.

Data. The data of τXBench is the result of applying a temporal simulation, τGENERATOR (Section 5.6), to the non-temporal XBench DC/SD dataset. This derived temporal dataset consists of books whose titles, authors, and publishers change over time, authors whose names and addresses change over time, and publishers whose lists of published books changes over time.

Workload. We defined a set of 85 τXQuery queries: a current and non-sequenced query for each of the original 17 queries from XBench DC/SD, and three sequenced queries for each of the original 17 XBench queries: a *short period* query that only considers 10% of the time line, a *medium period* query that considers 50%, and a *long period* query that considers 90%.

The queries themselves are the same as those in the XBench DC/SD workload, except with temporal statement modifiers prepended to the query. Semantically, the queries are expanded, but we prefer in the Benchmark-Relationship diagram to emphasize that an automated mapping (such as with TXL [12]) could convert the original queries into new queries by simply prepending the temporal statement modifiers. These new queries can then be executed by any tool that understands the statement modifiers, such as the τXQuery engine or by a source-to-source translator from τXQuery to XQuery.

5.3. *The* PSM *and* PSM-DB2 *Benchmarks*

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 [18, 39]. PSM consists of *stored procedures* and *stored functions*, which are collectively referred to as *stored routines*. The SQL/PSM standard [18] provides a set of control-flow constructs (*features*) for SQL routines.

We adopted the PSM benchmark from the XBench DC/SD benchmark to assess DBMSes that implement PSM functionality. The PSM-DB2 benchmark is a simple transformation of the schema and workload of the PSM benchmark so that the resulting benchmark is compatible and thus can be executed in IBM DB2.

Data. The data is reused as XBench DC/SD, and translated into six relations.

Shredding is the process of transforming an XML tree into a set of two dimensional DBMS relations [53]. 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 items and authors in the XBench DC/SD data. This relationship is implicitly captured in XML by making author a subelement of item, but must be explicitly captured in its own relation during the shredding process.

DOI: 10.1002/spe

We have created a tool to shred the XBench DC/SD data 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 element.

Schema. The schema is the same as XBench DC/SD, except expressed as tables in the relational DDL.

Workload. The workload is derived from the original 17 queries in the XBench DC/SD workload, translated to be in the form of relational DML (i.e., SQL/PSM). Six queries were omitted, since they had no conceptual counterpart in PSM, such as testing the access of relatively ordered XML elements. The workload was thus designed to be a microbenchmark, in that each PSM query highlights an individual construct of PSM, so that for example the performance of each construct can be studied in isolation, while retaining as much as possible the semantics of the original XBench query from which it was derived. Some queries, such as *q2* were also changed to highlight a different feature, such as multiple `SET` statements in *q2b* and nested `FETCH` statements in *q7c*. See also *q7b*. Also we added three queries *q17*, *q18*, and *q19*, and one variant, *q17b*.

- Query *q2* highlights the construct of `SET` with a `SELECT` row,
- Query *2b* highlights multiple `SET` statements,
- Query *q3* highlights a `RETURN` with a `SELECT` row,
- Query *q5* highlights a function in the `SELECT` statement,
- Query *q6* highlights the `CASE` statement,
- Query *q7* highlights the `WHILE` statement,
- Query *q7b* highlights the `REPEAT` statement,
- Query *q7c* highlights the nested `FETCH` statements,
- Query *q8* highlights a loop name with the `FOR` statement,
- Query *q9* highlights a `CALL` within a procedure,
- Query *q10* highlights an `IF` without a `CURSOR`,
- Query *q11* highlights creation of a temporary table,
- Query *q14* highlights a local cursor declaration with associated `FETCH`, `OPEN`, and `CLOSE` statements,
- Query *q17* highlights the `LEAVE` statement,
- Query *q17b* highlights a non-nested `FETCH` statement,
- Query *q18* highlights a function called in the `FROM` clause, and
- Query *q19* highlights a `SET` statement.

The result is a set of 17 queries.

The PSM-DB2 workload utilizes a source-to-source translation of standard PSM into the syntactic variant supported by IBM DB2.

5.4. The τPSM and Related Benchmarks

Temporal SQL/PSM is a temporal extension to SQL/PSM that supports temporal relations and queries in conjunction with PSM functionality [24, 56]. The approach is similar to τXQuery in that it requires minor new syntax beyond that already in SQL/Temporal to define temporal PSM routines. Temporal SQL/PSM enables current queries (which are evaluated at the current time), sequenced (which are evaluated logically at each point in time independently), and non-sequenced queries (which are the most general of temporal semantics), as well as these three variants of PSM routines, similar to those described earlier for τXQuery.

The τPSM benchmark was created to test and validate Temporal SQL/PSM. There have been two different time slicing techniques proposed for implementing Temporal SQL/PSM: *maximally-fragmented slicing* and *per-statement slicing* [24, 56]. These slicing techniques are each a source-to-source translation of the sequenced queries and PSM routines of the workloads into conventional queries and PSM routines that explicitly manage the timestamps in the data, to optimize the temporal queries in various ways.. Thus we created the τPSM-MaxMapped and τPSM-PerStmtMapped

DOI: 10.1002/spe

benchmarks for these techniques. We also used the translation discussed above to create the schemas and workloads for the two DB2 variants, so that the resulting queries can be executed in DB2. (We note that the SQL:2011 standard [59] and IBM DB2 10 very recently now support temporal tables; our transformations are on conventional, that is, non-temporal, tables and queries..)

Data. The data of τPSM is the result of translating (shredding) the temporal XML data of τXBench into six (temporal) relations (see the previous section). We defined three datasets of different temporal volatility: DS1, DS2, and DS3. DS1 contains weekly changes, thus it contains 104 slices over two years, with each item having the same probability of being changed. Each time step experiences a total of 240 changes; thus there are 25K changes in all. DS2 contains the same number of slices but with rows in related tables associated with particular items changed more often (using a Gaussian distribution), to simulate hot-spot items. DS3 returns to the uniform model for the related tuples to be changed, but the changes are carried out on a daily basis, or 693 slices in all, each with 240 changes, or 25K changes in all (the number of slices was chosen to render the same number of total changes). These datasets come in different sizes: SMALL (e.g., DS1.SMALL is 12MB in six tables), MEDIUM (34MB), and LARGE (260MB).

The data for the two τPSM-Mapped and two τPSM-Mapped-DB2 variants is equivalent to that of τPSM.

Schema. The τPSM schemas are derived from the schemas of PSM. 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, thereby applying independently at each point in time.

The schema for τPSM-Mapped is mapped from the temporal domain to the non-temporal domain (since DBMSes do not yet support the statement modifiers of τPSM), by removing `ALTER TABLE ADD VALIDTIME` keywords from the table creation commands.

The schema for τPSM-Mapped-DB2 is mapped to meet the syntax requirements of DB2.

Workload. We define 85 queries for τPSM that correspond to the 17 queries in the PSM benchmark: 51 sequenced queries (a short period, medium period, and long period sequenced query for each of the original 17), 17 non-sequenced, and 17 current. The queries are derived directly from the PSM queries by adding simple temporal statement modifiers (i.e., `VALIDTIME` for sequenced queries, `NONSEQUENCED VALIDTIME` for non-sequenced queries, and no change for current queries) to each of the queries.

The workload for four τPSM-Mapped variants are the result of an automated transformation, using the previously-mentioned Maximally-Fragmented and Per-Statement slicing techniques [24, 56], expressed in TXL, to create 170 queries in total.

5.5. The τXSchema Benchmark

τXSchema (Temporal XML Schema) is a language and set of tools that enable the construction and validation of temporal XML documents [14, 16, 33, 55, 63]. τXSchema extends XML Schema [40, 41] 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 τXSchema, any element type can be rendered as temporal by including in the schema 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. So a τXSchema document is just a conventional XML Schema document with a few annotations.

τXSchema 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

Prepared using speauth.cls
DOI: 10.1002/spe

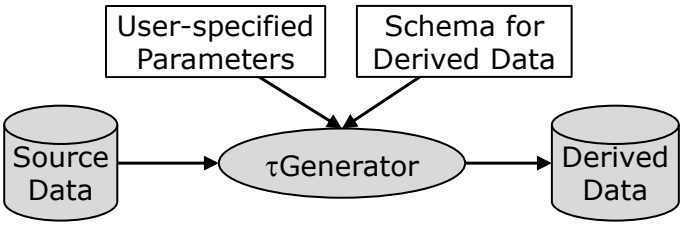

Figure 4. Overview of the τGENERATOR process.

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.

We have created the τXSchema benchmark to test the functionality and performance of τXSchema.

Schema. The schemas are the same as those in τXBench, augmented with seven additional manually-specified constraints. These constraints provide a simple but comprehensive set for testing the four types of XML constraints (referential integrity, data type, identity, and cardinality) with the various time modifiers (sequenced and non-sequenced). For example, one of the additional constraints specifies that "over a period of a year, an item may have up to 6 authors," exercising a non-sequenced cardinality constraint.

Data. The data is the same as the data in τXBench.

Workload. Because τXSchema is a schema language and tool system, and because the main goal of testing such a system is to test its schema constraints, there is no workload in this benchmark. (In effect, the schema *is* the workload, and a microbenchmark at that.)

5.6. Supporting Tools

To facilitate the creation and validation of the individual benchmarks in τBench, 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, along with the rest of τBench, is available online [64].

The message of this section is that tools are some of the most important glue connecting the benchmarks together into a framework. Some of the tools map a component, e.g., data or workload, in one benchmark into another, creating that component from hole cloth. Other tools perform consistency checks *within* a benchmark. Tools that perform cross-validation are especially critical, in that they ensure consistency between say the data components of two related benchmarks in the framework. Such validation leverage the mapping and within-benchmark consistency checking tools, because all the tools much work correctly for the end result to validate. These tools in concert thus tied the benchmarks together into a coherent whole, raising the quality of all of the individual benchmarks within the framework.

5.6.1. τGENERATOR τGENERATOR is a simulation that creates XML temporal data from the non-temporal XML data (Figure 4). To create the temporal data, τGENERATOR 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.

Simulation Description. The details of the simulation are described elsewhere [68]. Here we give only a brief overview of the issues involved [63].

We designate the `<item>`, `<author>`, `<publisher>`, and `<related_items>` elements of the DC/SD XBench XML data to be *temporal elements*, that is, their values can change over time. The

DOI: 10.1002/spe

overall goal of the simulation is to randomly change, at specified time intervals, the content of these four temporal elements. The simulation consists of the following four steps.

1. *Setup.* The simulation reads the user-created parameters file to initialize the simulation parameters. The simulation then uses DOM [72] to load the user-specified XBench document (in this case, `catalog.xml`) into memory. The simulation then creates an internal representation of the temporal items (all instances of the four temporal elements), which at this point consist of only a single version.

2. *Initial snapshot selection.* The simulation selects a user-specified percentage of `<item>` temporal elements (and all of their sub-elements) for the initial slice. The simulation selects the elements uniformly at random from `catalog.xml`. Elements that are selected are called active; those that are not selected for the initial slice are called inactive and put into a pool for later use.

The simulation sets the begin and end time of the first (and only, at this point) version of each selected temporal element to the user-specified begin date and the user-specified forever date, respectively.

3. *Time step loop.* The simulation increases the current time by the user-specified time step. The simulation then:

- Randomly selects a set of active elements to change, based on user-specified parameters.
- Changes the selected elements, as well as adds new elements and deletes existing elements, based on user-specified parameters. New elements, and the new values for changed elements, come from the pool of inactive elements.
- Outputs the current slice in XML format. A temporal document, i.e., all slices merged together, is also maintained.
- Checks the stop criteria of the simulation. If the simulation has exceeded the user-specified number of necessary changes, the simulation exits the time step loop and continues to Step 4. Otherwise, it returns to Step 3.

4. *Output.* The simulation outputs the final temporal document.

During the above simulation process, it is important to maintain the original constraints defined and implied in the DC/SD 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 `id`s of its own—all items keep their originally-assigned `id`s. Thus, as long as the original data is correct, this portion of the generated data will also be correct.

However, we do need to consider the referential integrity constraint between item `id`s and related items. Since only a subset of items are active at any given time, there is no guarantee that related items will point to active item `id`s. Thus, the simulation must provide this guarantee explicitly. The simulation satisfies this by adding a check at the end of each time step. The check builds a list of all active item `id`s. Then, for any currently active related items that refer to an 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 item `id`s and related items.

We also need to consider the cardinality constraints placed on authors. Since the simulation has the ability to add new authors and delete existing authors at any point in time, the simulation must be careful not to violate the `maxOccurs` and `minOccurs` constraints in the original schema. This is true for both the sequenced (i.e., at any given time, an item must have between 1 and 4 authors) and non-sequenced (i.e., in any given year, an item can have up to 6 authors) variants of this constraint. The simulation achieves these by maintaining an author counter for each item and each time period. For the sequenced constraint, the counter is incremented and decremented as the simulation adds or removes authors. For the non-sequenced, the counter is only incremented when an author is inserted. Finally, when an item is selected by the simulation to insert or remove an author, the simulation first consults this counter; if the counter indicates that such an action will

DOI: 10.1002/spe

violate a constraint, the simulation chooses another 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.

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

5.6.2. Shredding XML Data into Relations As mentioned above, we have created a tool to shred the XBench DC/SD data into six relations. If the data to be shredded is temporal, the resulting relations will have two additional columns representing the begin and end time for each row.

5.6.3. Validation Tools We have developed several tools to help us validate our translations and simulations.

Validating the Output of τGENERATOR. We have created a tool which validates the XML slices generated by τGENERATOR against their XML schema using XMLLINT [36]. In addition, the tool validates the generated temporal document against its temporal schema using τXMLLINT [14]. 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 and non-sequenced semantics of the conventional schema.

Corrupting Data. Constraints should be tested on both valid and invalid data. We thus corrupt the data generated by τGENERATOR as a testing phase to determine if the validators and associated schemas can detect the corruptions. We have developed a tool, called τCORRUPTOR, that 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 modified to directly contradict one of the constraints specified in the original schema.

Specifically, the input into τCORRUPTOR is a temporal XML document generated by τGENERATOR, as well as a integer parameter specifying which of the seven temporal constraints to invalidate.

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` attribute to a random string.
3. Sequenced identity. Copies the `ISBN` of one item to another.
4. Sequenced datatype. Changes the number of pages to a random string.
5. Non-sequenced identity. Copies the `id` of an item i_1 at time t_1 to the `id` of an item i_2 at time t_2, where $i_1 \neq i_2$ and $t_1 \neq t_2$.
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).

Checking Primary Keys of Shredded Relations. We have created a tool that checks each shredded relation to ensure that each primary key is unique across the relation. It does so by building a hash table of primary keys in each relation. If a row is encountered whose key is already defined in the hash table, then the tool reports an error. Relations whose primary key are defined across multiple columns require multi-dimensional hash tables.

Checking Referential Integrity of Shredded Relations. We have created a tool that 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.

Comparing XML Slices to the Temporal XML Document. We have created a tool that ensures that the generated temporal XML document and the corresponding XML slices are consistent. The

DOI: 10.1002/spe

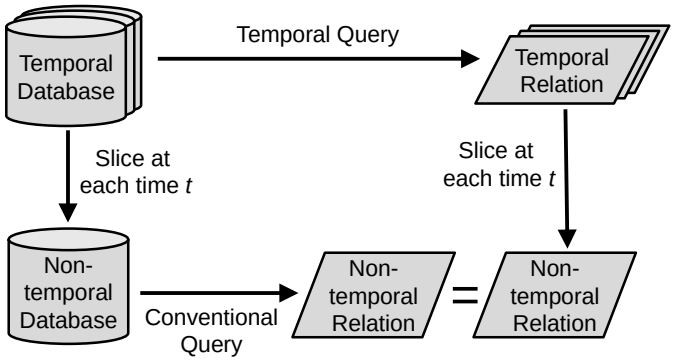

Figure 5. Commutativity validation.

last tool makes use of UNSQUASH, which is part of the τXSchema tool suite [14]. UNSQUASH extracts each XML slice from a given temporal XML document. Our tool then compares each generated slice with each extracted slice using X-Diff [75] to ensure that they are equivalent. If all the slices are equivalent, then our tool reports that the temporal XML document and generated slices are equivalent.

Testing Queries. *Query Executor* is a tool providing a simple way to quickly execute multiple queries for a variety of DBMSes. After the DBMS and password is entered the GUI will display a list of active query directories and allow for additional directories to be added, or for old directories to be removed. Each query directory contains a parameters file (`.queryexecutor`) that specifies a connection to the appropriate database. Queries within the selected directory appear in a separate pane and the tool allows a query to be run individually, all of the queries within the directory to be run, or a selected group of the queries to be executed. The results of each query are stored in files and Query Executor can run a diff on the last two executions. If one or more of the previous two runs was of more than a single query the diff will be performed only on the queries from both runs that have the same query number. Query Executor also allows for the results of any run or diff to be exported to a file. This tool has proven very convenient for developing the queries within each benchmark.

Comparing XQuery and PSM Queries. We have created a tool that performs a pair of 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 different, the tool reformats the PSM output to be of the same format as the XQuery output. If the two results are textually identical, then the queries achieved the same functionality.

This tool is part of our larger commutativity validation process. *Commutativity validation* compares, for example, the output of performing a temporal query on a temporal database with the output of performing a conventional query on a slice of the temporal database (Figure 5). Such a comparison simultaneously validates several tasks and tools: slicing the temporal database; executing temporal queries; slicing temporal relations; and executing conventional queries. Only if all of these steps are working perfectly will the results be the same, thus a strong level of validation is ensured. This cross-validation relies on snapshot equivalence and temporal upward compatibility [5], as well as the sequenced semantics [8, 54], which relate in very simple ways non-temporal and temporal data, queries, and constraints.

5.6.4. Example: Creating the τXBench Benchmark In order to illustrate the procedure (and power) of benchmark frameworks, we now present a detailed example from τBench. We show how we created the τXBench benchmark from the DC/SD XBench benchmark.

First, we execute τGENERATOR in order to create the temporal data. We specify an arbitrary starting time of January 2010. We then specify that we want a new slice every week, for a total of

DOI: 10.1002/spe

104 weeks (two years). At each new slice, 240 elements are added, changed, or deleted; thus there are 25K changes in all. τGENERATOR performs the simulation and outputs the final temporal XML document, `temporal.xml` (34MB).

τGENERATOR also creates a temporal schema, `temporalSchema.xml`, which simply points to the conventional XML schema and to the temporal document. It also contains a couple of simple temporal annotations, specifying, for example, that the `<item>`, `<author>`, `<publisher>`, and `<related_items>` elements are temporal elements; that each temporal element has a unique `id`; and that each `<related_item>` points to an existing (and active) `<item>` `id`.

Then, we use our tools to validate the generated output.

Next, we create temporal queries from DC/SD XBench's non-temporal queries simply by adding statement modifiers (i.e., current and sequenced); adding no modifier implies non-sequenced semantics. We validate the results of the current queries by comparing their output against the output of the original non-temporal queries executed on a slice of the `temporalSchema.xml` (using UNSQUASH) at time *now*. We validate the output of the sequenced queries using the commutativity tool. We do not validate the non-sequenced queries, as they are simply returning the results of the temporal simulation.

After all of these steps, we can be confident that data is correct, the schema is correct, and the workload is correct. In addition, we know (and can easily describe) the exact origins of the data, schema, and workload.

5.6.5. Running the Benchmarks We have developed a tool, QUERYRUNNER, that presents a graphical interface to run a specified set of queries over a specified data set, collecting the relevant metrics. This tool is applicable to the benchmarks utilizing the relational model.

5.7. Summary and Lessons Learned

In creating τBench, several tools and concepts were realized. The original XBench benchmark was branched in two different directions: creating a temporal version of XBench (τXBench), and creating a PSM version of XBench (PSM). Each of these were further branched for additional purposes. To do so, we created tools to vary a nontemporal dataset over time, to change hierarchical data into relational data, to validate the data changes, and many other tasks. The resultant benchmarks, their relationships, and the tools used to create them, are all a part of τBench.

As we built τBench, we were surprised by the number of subtle bugs that we found in the temporal data generator (τGENERATOR), which we only discovered by validating and executing the derived benchmarks. With each consistency and validation check that we added to the benchmarks, more bugs were uncovered. In the end, we developed confidence that τGENERATOR, and hence our datasets, was correct and consistent. This process highlights the strength of the component connections, and their ability to be cross checked, of benchmark frameworks.

We found that each additional benchmark required less time to create than the previous. Instead of having to create ten benchmarks from scratch, we created a single benchmark and slightly modified it ten times, using tools to automate as much as possible. In practice, we use each benchmark in τBench for some research project, and we have already reaped the benefits of reuse. in terms of reduced development cost. We feel that benchmark frameworks are well worth any initial start-up costs.

In addition, we have found that describing the benchmarks has been simplified, due to the added organizational clarity provided by regularized benchmark components and the B-R diagrams.

Finally, we found that since each benchmark component tends to be re-used by derived benchmarks, it becomes easier to justify the extra effort of ensuring each component is of the highest quality.

Our experience with developing τBench has been that each aspect of a benchmark framework, including explicating shared and changed and new components in a new benchmark, exploiting the opportunity for cross validation, and sharing data and tools across constituent benchmarks, brought with it benefits that fully justified the small amount of effort and rigor required. And sometimes,

DOI: 10.1002/spe

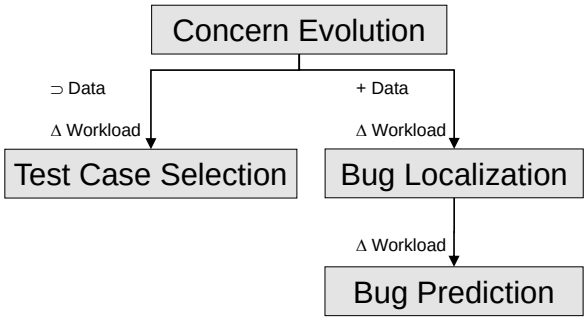

Figure 6. B-R diagram of the MUD benchmark framework.

the sharing of components made developing a descendant benchmark substantially easier than developing one from whole cloth.

To further demonstrate the generality and usefulness of benchmark frameworks, we now briefly present two additional case studies from different domains: the MUD framework and the TPC framework, examining how the general principles listed in Section 4.1 apply in these domains.

6. THE MUD BENCHMARK FRAMEWORK

Mining Unstructured Data (*MUD*) is a project from the empirical software engineering domain that uses information retrieval (IR) models to mine the unstructured data found in software repositories such as developer emails, bug report titles, source code identifiers and comments, and requirements documents [65]. An ongoing research project by the first author, the MUD project considers different software engineering tasks, such as test case selection and bug localization, with benchmarks for each. We have created a benchmark framework to describe the MUD project, which we now present.

The organizational insight of the MUD benchmark framework is sharing (i) the data model, (ii) DDL, (iii) data, (iv) data preprocessing techniques, and (v) validation tools across the constituent benchmarks, to enable a common approach for evaluating different algorithm variations.

Figure 6 shows the Benchmark-Relationship diagram for the MUD benchmark framework, which consists of four constituent benchmarks.

6.1. The Concern Evolution *Benchmark*

The foundation of the framework is the Concern Evolution benchmark, initially created for measuring how the *concerns* (i.e., technical aspects, such as "printing", "database connections", "logging", and "XML I/O") in a software system [47] are evolving over time [66]. Here, the data component is the source code of several real-world software projects (Eclipse JDT, Mozilla, Chrome, and AspectJ). The source code is not executed by this benchmark, but instead is treated as simple text and input into different text mining algorithms. Specifically, the identifier names, comments, and string literals from the source code are used, as these elements have been shown to represent the developer's thoughts and intentions of the code [2], making them appropriate for automated concern mining algorithms. These code elements can be subjected to different preprocessing steps, such as splitting identifier names (camelCase and under_scores) and removing stop words. Hence, the dataset does not contain a set of executable commands, but rather a set of words that happen to be extracted from source code. The workload consists of the concern evolution mining algorithm variations being researched, as algorithms can vary in which preprocessing steps are performed and in which concern mining technique is used, such as LDA [7] or DiffLDA [67]). The data model and DDL are both the Java programming language, as the data is in the form of Java source code. The schema contains informally-expressed constraints, such as "identifiers have no embedded spaces". The DML is the R programming environment, since that is how the algorithm variations are implemented.

DOI: 10.1002/spe

6.2. *The* Test Case Collection *Benchmark*

The Test Case Selection benchmark is useful for evaluating information retrieval-based test case selection algorithms, which aim to choose a subset of all available test cases while still detecting as many faults in the software as possible [48]. IR-based test case selection algorithms take as input a set of test cases (i.e., the source code of the test cases), and output a subset of test cases (usually much smaller than the total set) which, when executed, will still likely detect a majority of the bugs in the system. The selection algorithms work by maximizing the distance (in this case, using textual difference metrics, such as the Levenshtein distance, between the source code of the test cases) between the subset of selected test cases, thereby ensuring that all parts of the underlying software system are tested. Here, the workload consists of the selection algorithm variations (i.e., different preprocessing steps, different distance metrics, and different maximization algorithms), and the data is only a subset of the data from the Concern Evolution benchmark, namely the source code files that correspond to test cases. All other components remain the same.

6.3. *The* Bug Localization *Benchmark*

IR-based bug localization algorithms [37], which aim to identify which source code files likely contain faults (i.e., bugs), given a bug report from a bug repository (e.g., Bugzilla [42]). These algorithms work by building an index on the (preprocessed) source code, and then query this index using the content of the bug reports. Again, the workload is the set of bug localization algorithm variations (i.e., different IR models, different similarity measures, and different ranking algorithms). The data is expanded from the Concern Evolution data to include bug report data, such as the bug report title and detailed description entered by user; all other components remain the same.

6.4. *The* Bug Prediction *Benchmark*

Finally, the Bug Prediction benchmark is used to evaluate bug prediction algorithms, whose aim is to predict the location of bugs in the source code, this time without the help from a bug report [17]. Bug prediction algorithms work by measuring metrics on each source code module (e.g., complexity, coupling/cohesion with other modules, number of changes, number of previous bugs), and train statistical models to predict the likelihood that each module contains a bug. Here, the workload consists of bug prediction algorithm variations (i.e., which metrics to measure, what statistical models to use), but all other components remain the same.

6.5. *Supporting Tools*

In the creation, validation, and execution of the MUD benchmark framework, the following tools are shared (reused).

Data generation. The data for the original Concern Evolution benchmark is comprised of the source code files of several real-world systems, totaling over 70 million lines of code. Specifically, the data contains the source code of Eclipse (the 16 official releases from 2002–2009), IBM Jazz (4 releases from 2007–2008), and Mozilla (10 release from 2004–2006). Typically, when using information retrieval or other text mining algorithms on source code, as MUD does, the source code is first preprocessed to capture the domain-specific information encoded in the identifier names, comments, and string literals in the code. The data generation tool for MUD thus preprocesses each source code file to isolate these elements, removing language-specific syntax, keywords, etc. Additionally, the data generation tool splits identifier names, removes common English language stop words, and stems each word in an effort to normalize the vocabulary (Figure 7).

Validation Tools. Many validation tools are used to check the consistency of the data and data generation tools, as well as the results of the different workloads. For example, the lightweight source code preprocessor (LSCP) tool has internal validation capabilities to ensure that the data preprocessing is performed correctly, by checking the associativity of the tool. (For example, the tool is run three times: once to isolate identifiers only; once to isolate comments only; and once

DOI: 10.1002/spe

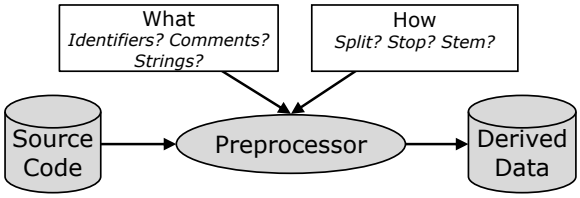

Figure 7. MUD data generation.

to isolate comments and identifiers. LSCP then compares the union of the first two runs with the third.) Similar checks can be performed for other tool parameters.

Additionally, many of the benchmarks train IR models on the data, which have their own class of validity checks. For example, the latent Dirichlet allocation (LDA) [7] model assigns topics to each document; we have a validation script in R to check that the sum of each document's assigned topics equals one, by the definition of the LDA model. Dozens of similar checks are performed to ensure the correctness of the data.

6.6. Summary and Lessons Learned

The MUD benchmark framework is simpler than τBench, but many of the ideas are the same. Starting with a foundation benchmark (i.e., Concern Evolution), the benchmark components are changed to create new benchmarks. In the case of MUD, the data and workload components were changed to create the Test Case Selection and Bug Localization benchmarks; the Bug Localization benchmark was further changed to create the Bug Prediction benchmark. Tools were created to automate the generation (i.e., via preprocessing) of the data and to validate the correctness of several IR models. The resultant benchmarks, their relationships, and the data creation and validation tools are all a part of MUD.

During the creation of MUD, we learned a few important lessons. The first was the power of regularizing the dataset in the root benchmark, so that it can easily be reused in subsequent benchmarks. For example, when starting the bug localization project, instead of having to hunt for appropriate data and create a new benchmark from scratch, we simply re-used the data from the Concern Evolution benchmark, added the bug reports, and implemented the algorithm variations.

As was the case with τBench, we also found in MUD that the consistency and validation checks helped to find subtle bugs and corner cases in the data generator. For example, a step in one of the text mining algorithms in the workload of the Bug Localization benchmark is supposed to output a vector which sums to one for every input document in the data. However, due to an inconsistent parameter setting in the algorithm, the algorithm produced vectors containing all zeroes for some input documents. Since we had created a data validation tool to check the vectors for this, we were able to easily uncover and correct this error.

Finally, we found that by describing the research projects as benchmarks, which consist of course of data and the workloads applied to the data, the resulting organizational clarity enable us to more easily implement and test new algorithm variants. For example, with the Bug Localization benchmark in place, a researcher can test a new suite of bug localization algorithms by simply deriving a new benchmark and replacing the workload, leaving the other components the same.

7. TPC: A REPRISE

As mentioned in the introduction, the TPC family of benchmarks, while loosely related, presents complexity for end users in exercising these benchmarks in a unified fashion, due to the lack of explicit relationships. We present a *proposed* benchmark framework for TPC to illustrate the advantages of benchmark frameworks. We propose the organizational insight is structuring the constituent benchmarks around a single relational database that is both realistic and appropriate for both transactional and analytical applications.

DOI: 10.1002/spe

Figure 8 shows our TPC benchmark framework, comprised of four individual benchmarks, along with two separate benchmarks derived from them.

The TPC-C benchmark contains a large dataset for a wholesale supplier, consisting of warehouses, sales districts, and customer transactions. The OLTP workload consists of mostly simple queries as well as small modifications such as `INSERT` and `UPDATE` operations.

The TPC-H benchmark utilizes a very similar, but not identical, set of tables. This benchmark could in fact have shared the same data and schema as TPC-C. But because TPC-H emphasizes analytical applications, it uses an entirely new OLAP workload, consisting of sophisticated analytical queries that extensively involve joins, predicates, and aggregations. For that reason, in our proposed framework, we show the only difference between the two benchmarks being a changed workload.

The TPC-E benchmark was designed to evaluate DBMSes with a more generalized workload, that of complicated analytical queries with transaction processing. As a family, this benchmark differs in small but non-essential ways from both of the prior benchmarks. In our proposed framework, TPC-E is simply a merger of the TPC-C and TPC-H benchmarks, containing the same data and schema as both, with a union of their queries.

The TPC-R benchmark is the same as TPC-H, except allows for optimizations to be performed before executing the queries, such as creating indices on certain columns to enhance average query performance. Thus, all benchmark components are the same, except the schema (which now contains `CREATE INDEX` statements).

Even though this benchmark framework is simple, its understandability has been greatly increased due to greater organizational clarity. (A B-R diagram of the actual TPC family of benchmarks would be quite a bit more complex, because of superfluous changes of components between benchmarks.) For example, an end-user may not have previously known if the TPC-C and TPC-R benchmarks could have used the exact same data; now, it is clear that they do. And tools, such as those for generating sample data at various scale factors and for executing the workload against the database and collecting the metrics, could be reused across the framework. Finally, the workload, data, and DBMS query processing in the TPC-H, TPC-E and TPC-R benchmarks can be cross-validated by ensuring that the queries return the exact same results in these three benchmarks.

Moreover, we found that in practice, such benchmarks are often modified for evaluating certain features of newly developed DBMSes. For instance, Stonebraker et. al. investigated a column-oriented DBMS implementation, namely the *C-Store* [60]. They evaluated this system with a modified version of the TPC-H benchmark in which five tables were removed and the schema of the remaining three tables were modified. In addition, they modified one original query from the benchmark and added six new queries. Another example is the evaluation of the *MonetDB/X100* system performed by Boncz et. al., in which the schema of the TPC-H benchmark is modified by creating additional indices on specific columns and sorting particular tables (both of which are additional physical schema additions) before queries are run [10]. A third example is a version of TPC-C used to evaluate the automated partitioning facilities in the *Horticulture* system, in which they "generated a temporally skewed load for TPC-C, where the `WAREHOUSE_id` used in the transactions' input parameters is chosen...[to present] a significant amount of skew" [46].

If the TPC benchmarks employed the benchmark framework approach, these specific modifications performed by the above discussed research could be more explicitly depicted in the framework's B-R diagram. Moreover, the change(s) applied on each path (e.g., data or schema) of deriving a new benchmark from an existing one can be more clearly communicated. The TPC framework B-R diagram emphasizes that the TPC-H CStore benchmark is in fact changed in both schema and workload, making it clear that the experimental results depend in part on these changes and thus may not be applicable to the original TPC-H benchmark. This diagram also makes explicit that the TPC-C (Skewed) benchmark has a skewed *workload*.

DOI: 10.1002/spe

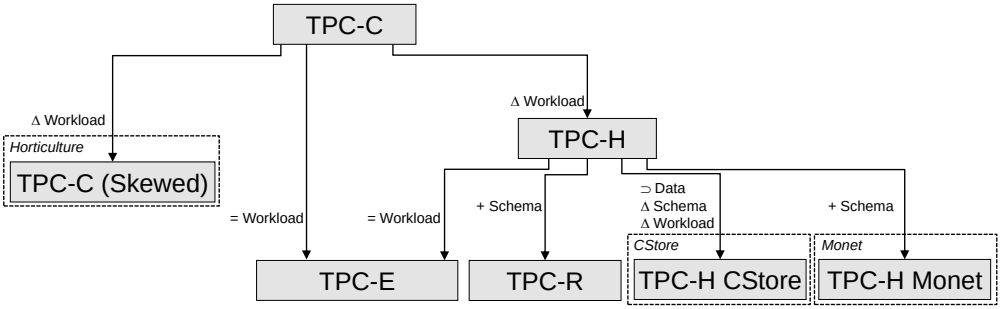

Figure 8. B-R diagram of a proposed TPC Framework.

8. ADVANTAGES AND DISADVANTAGES OF BENCHMARK FRAMEWORKS

Benchmark frameworks are not meant to suite every purpose and every benchmark. When commercial politics mandate a specific benchmark to be used time and time again, without modifications to the benchmark, benchmark frameworks will offer no advantages.

The initial startup cost of benchmark frameworks (i.e., clearly defining benchmark components and their relationships, creating shared tools and consistency checks) may be high for small families of simple benchmarks, or when the benchmarks share little-to-no components (in which case, it is probably best to merge such benchmarks into neither a family nor a framework).

At the same time, the organization and mechanics of benchmark frameworks bring many benefits to both the developers and the users of the individual benchmarks. First, benchmark frameworks encourage the reuse of existing benchmarks and their components, without the need of creating new components from scratch. Reuse is especially helpful with large and complex datasets, schemas, and workloads.

Reuse also brings forth another advantage, that of reducing the development cost of subsequent benchmarks. Components can be reused and iteratively modified to suite the new benchmark's purposes. Researchers and practitioners who may have otherwise not created a new benchmark now enjoy a lower development cost. High quality need not be traded off against greater development effort: the former can be increased at the same time that the latter is decreased through the general principles inherent in benchmark frameworks.

The structure and organization of benchmark frameworks, embodied by the B-R diagrams and explicit connections between components, are easier to understand than traditional benchmark families. When large, distributed groups collaborate to develop related benchmarks, the organization provided by benchmark frameworks becomes especially important and useful.

As illustrated in Section 7, the B-R diagram emphasizes what is borrowed and what is changed when a benchmark is not used wholesale, but rather piecemeal in the evaluation of a software or hardware system.

Finally, and perhaps most importantly, the structuring of the framework and the sharing and cross validation that this structure encourages results in constituent benchmarks whose correctness is more demonstrable, yielding greater confidence on the part of the developers and users of the correctness of those constituent benchmarks. For benchmarks that are to be actually used for evaluation, the relatively small amount of effort required to create a benchmark framework is generally well worth this added confidence. As evidence, we have already had interest from benchmarking practitioners involved with the TPC family, due to the benefits they see in the framework approach.

9. CONCLUSIONS AND FUTURE WORK

We have introduced the notion of a benchmark framework, which is an ecosystem of related benchmarks having well-defined and well-specified relationships, as well as tools to help transform

DOI: 10.1002/spe

and validate individual benchmarks, based on organizational principles. These synergistic properties of benchmark frameworks help overcome the difficulties of maintaining, describing, and validating families of related benchmarks. Benchmark frameworks bring *clear communication* for describing the changes made to existing benchmark components, by using Benchmark Relationship diagrams. Benchmark frameworks also encourage *sharing* and *reusing*, allowing an existing benchmark to be quickly adapted to test new systems or strategies, by using automated tools to translate existing components into new ones. Benchmark frameworks encourage *regularization of the execution* of individual benchmarks, by using shared tools and environments. Finally, benchmark frameworks help ensure *validity*, by consistently evaluating the validity of benchmark components from various angles (e.g., commutativity).

We have also introduced τBench, a benchmark framework for evaluating temporal schema and query languages on XML and relational data [68]. Although τBench spans many technologies (from XML Schema to Temporal SQL/PSM) and is used by several distinct projects, the benchmarks are easy to understand and creating new, related benchmarks is straightforward and reliable; this is one of the powers of benchmark frameworks. An early implementation of τBench is available on our project website [64]. Our experience with this framework is that it was *easier* to develop than a family of only loosely related benchmarks, due to extensive sharing of components and tools between constituent benchmarks.

To emphasize the generality of benchmark frameworks, we also briefly presented two additional benchmark frameworks. Although these were from different domains (those of empirical software engineering and OLTP/OLAP), the advantages of benchmark frameworks were still available: relationships between benchmarks were explicit and tools helped in the creation and validation of each individual benchmark.

In future work, we would like to augment the relationship constraints in Table II. We have developed an XML schema for Benchmark-Relationship diagrams; it would be useful to have a meta-tool to test such constraints. The meta-tool can check constraints by reading the data, schemas, and workloads of two benchmarks and ensuring that they comply with the stated relationships in the Benchmark-Relationship diagram. For example, if the diagram shows that the data component of one benchmark is equivalent to the data component of its parent benchmark, then the meta-tool can ensure this equality.

In this paper, we have discussed the generality of benchmark frameworks, and the methods to create and validate them. In future work, as we and others develop more benchmark frameworks from other domains, we wish to determine whether the tools could themselves also benefit from general principals. For example, can we find general design choices and techniques for data generation and validation tools, no matter the underlying domain of the benchmark framework? Or are these tools only contributory to such general principals?

The τBench benchmark framework can be expanded in several directions: automating some of the mappings (such as for the τXBench workload), adding new DBMSes for PSM (that should be easier with automated mappings), adding new mapping strategies for τPSM(also aided by automated mappings), adding new schema languages for XML, and adding new implementation strategies for τXSchema. In particular, several DBMSes now offer temporal support. Oracle 10g added support for valid-time tables and transaction-time tables, and Oracle 11g enhanced support for valid-time queries [45]. Teradata recently announced support in Teradata Database 13.10 of most of these facilities as well [62], as did IBM for DB2 10 for z/OS [30].

Designing an individual benchmark as well as a realized benchmark framework is the result of a long sequence of design decisions. For example, in designing the PSM benchmark, we adopted a particular way of shredding the XML document into relations. It would be useful to have a way of capturing such inter-benchmark transformations in a more complete manner than just the five relationships listed in Section 4.2, for example with annotations that name the software that performs that shredding, or better yet, with a formal notation that gives the semantics of that transformation. Doing so would richen the kinds of relationship constraints (e.g., Table II) that could be stated and verified.

DOI: 10.1002/spe

Finally, we encourage the XBench, SPEC and TPC organizations to consider refining their respective benchmark families into more structured and more convenient benchmark frameworks.

ACKNOWLEDGMENTS

This research was supported in part by NSF grants IIS-0415101, IIS-0639106, IIS-0803229, IIS-1016205, and a grant from Microsoft. Jennifer Dempsey helped develop τBench, including the Query Executor tool. We thank the referees for their helpful comments, which improved the paper.

REFERENCES

1. L. Afanasiev, I. Manolescu, and P. Michiels. Member: A micro-benchmark repository for xquery. In S. Bressan, S. Ceri, E. Hunt, Z. Ives, Z. Bellahsne, M. Rys, and R. Unland, editors, *Database and XML Technologies*, volume 3671 of *Lecture Notes in Computer Science*, pages 578–578. Springer Berlin / Heidelberg, 2005. 10.1007/11547273_11.
2. G. Antoniol, Y. G. Gueheneuc, E. Merlo, and P. Tonella. Mining the lexicon used by programmers during sofware evolution. In *International Conference on Software Maintenance*, pages 14–23, 2007.
3. Apache. Log4J: Logging services. `http://logging.apache.org/log4j/1.2/`, viewed November 25, 2011.
4. M. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., 1998.
5. J. Bair, M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Notions of upward compatibility of temporal query languages. *Business Informatics (Wirtschafts Informatik)*, 39(1):25–34, 1997.
6. D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *Proceedings of the International Conference on Management of Data*, pages 616–616, 2002.
7. D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
8. M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *ACM Transactions on Database Systems*, 25(4):407–456, 2000.
9. T. Böhme and E. Rahm. Xmach-1: A benchmark for xml data management. In *Proceedings of German database conference BTW2001*, pages 264–273, March 2001.
10. P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovations in Database Research*, pages 225–237, 2005.
11. S. Bressan, M. Li Lee, Y. Guang Li, Z. Lacroix, and U. Nambiar. The XOO7 benchmark. *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web*, pages 146–147, 2003.
12. J. Cordy, C. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.
13. M. Corporation. WSTest web services benchmark. `http://msdn.microsoft.com/en-us/netframework/cc302396`, viewed November 25, 2011.
14. F. Currim, S. Currim, C. Dyreson, S. Joshi, R. T. Snodgrass, S. W. Thomas, and E. Roeder. τXSchema: Support for data- and schema-versioned XML documents. Technical Report TR-91, TimeCenter, 2009.
15. F. Currim, S. Currim, C. Dyreson, R. T. Snodgrass, S. W. Thomas, and R. Zhang. Adding temporal constraints to XML schema. *IEEE Transactions on Knowledge and Data Engineering*, 24:1361–1377, August 2012.
16. F. Currim, S. Currim, C. E. Dyreson, and R. T. Snodgrass. A tale of two schemas: Creating a temporal XML schema from a snapshot schema with τXSchema. In *Proceedings of the 9th International Conference on Extending Database Technology*, pages 348–365, 2004.
17. M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4):531–577, 2012.
18. C. Date and H. Darwen. *A guide to the SQL standard*. Addison-Wesley New York, 1987.
19. M. Fayad and D. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.
20. P. Fischer. XQBench, 2011. `http://xqbench.org/`, viewed November 25, 2011.
21. M. Franceschet. Xpathmark - an xpath benchmark for xmark generated data. In *International XML Database Symposium (XSYM)*, pages 129–143, 2005.
22. Free Software Foundation. GNU General Public License, 2010. `http://www.gnu.org/licenses`, viewed October 2, 2010.
23. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA, 1995.
24. D. Gao. *Supporting the procedural component of query languages over time-varying data*. PhD thesis, University of Arizona, 2009.
25. D. Gao and R. T. Snodgrass. Syntax, semantics, and evaluation in the τxquery temporal XML query language. Technical Report TR-72, TimeCenter, 2003.
26. D. Gao and R. T. Snodgrass. Temporal slicing in the evaluation of XML queries. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 632–643. VLDB Endowment, 2003.
27. J. Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

DOI: 10.1002/spe

28. R. W. Grosse-Kunstleve, N. K. Sauter, N. W. Moriarty, and P. D. Adams. The computational crystallography toolbox: Crystallographic algorithms in a reusable software framework. *Journal of Applied Crystallography*, 35:126–136, 2002.
29. J. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.
30. IBM Corporation. A Matter of Time: Temporal Data Management in DB2 for z/OS, Dec. 2010. accessed December 2010.
31. INEX. Initiative for the Evaluation of XML retrieval. `https://inex.mmci.uni-saarland.de/`, viewed July 14, 2011.
32. R. Johnson. Frameworks=(components+ patterns). *Communications of the ACM*, 40(10):39–42, 1997.
33. S. Joshi. τXSchema - Support for data- and schema-versioned XML documents. Master's thesis, Computer Science Department, University of Arizona, August 2007.
34. A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *Proceedings of the 10th Conference on Internet Measurement*, pages 1–14, 2010.
35. Y. Li, S. Bressan, G. Dobbie, Z. Lacroix, M. Lee, U. Nambiar, and B. Wadhwa. XOO7: Applying OO7 benchmark to XML query processing tool. In *Proceedings of the 10th International Conference on Information and Knowledge Management*, pages 167–174, 2001.
36. Libxml. The XML C parser and toolkit of Gnome, version 2.7.2, 2008. `http://xmlsoft.org/`, viewed February 5, 2009.
37. S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.
38. L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.
39. J. Melton. Understanding SQL's stored procedures: A complete guide to SQL/PSM. Morgan Kaufmann Publishers, 1998.
40. N. Mendelsohn. XML Schema part 1: Structures second edition, October 2004. `http://www.w3.org/TR/2001/REC-xmlschema-1/`, viewed November 25, 2011.
41. N. Mendelsohn. XML Schema part 2: Datatypes second edition, October 2004. `http://www.w3.org/TR/2001/REC-xmlschema-2`, viewed November 25, 2011.
42. Mozilla Foundation. Bugzilla. http://www.bugzilla.org/, 2011.
43. M. Naftalin and P. Wadler. *Java generics and collections*. O'Reilly Media, Inc., 2006.
44. M. Nicola, I. Kogan, and B. Schiefer. An xml transaction processing benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 937–948, New York, NY, USA, 2007. ACM.
45. Oracle Corporation. Workspace Manager Developer's Guide 11g Release 1 (11.1), Aug. 2008.
46. A. Pavlo, E. P. C. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *Proc. VLDB Endow.*, 5(2):85–96, Oct. 2011.
47. M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1), Feb. 2007.
48. G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Transactions on Software Engineering*, pages 929–948, 2001.
49. K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa. The Michigan benchmark: towards XML query performance diagnostics. *Information Systems*, 31:73–97, 2006.
50. A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 974–985, 2002.
51. A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *International Conference on Very Large Data Bases (VLDB)*, pages 974–985, August 2002.
52. S. A. Schneider, M. A. Ullman, and V. W. Chen. Controlshell: A real-time software framework. In *Prceedings of the International Conference on Systems Engineering*, pages 129–134, 1991.
53. J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, E. Viglas, J. Naughton, and I. Tatarinov. A general technique for querying XML documents using a relational database system. *SIGMOD Record*, 30(3):20–26, 2001.
54. R. T. Snodgrass. *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2000.
55. R. T. Snodgrass, C. Dyreson, F. Currim, S. Currim, and S. Joshi. Validating quicksand: Temporal schema versioning in τXSchema. *Data Knowledge and Engineering*, 65(2):223–242, 2008.
56. R. T. Snodgrass, D. Gao, R. Zhang, and S. W. Thomas. Temporal support for persistent stored modules. In *Proceedings of the 28th International Conference on Data Engineering, to appear*, April 2012.
57. SPEC. Standard Performance Evaluation Corporation. `http://www.spec.org/benchmarks.html`, viewed July 14, 2011.
58. SPEC. SPEC Organizational Information, URL `http://www.spec.org/spec`, 2011. Viewed March 12, 2011.
59. SQL Standards. Part 2: Foundation. Technical Report ISO/IEC 9075-2:2011, December 2011.
60. M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-Oriented DBMS. In *Proceedings of the International Conference on Very Large Databases*, pages 553–564. VLDB Endowment, 2005.
61. M. Sturm, A. Bertsch, C. Gröpel, A. Hildebrandt, R. Hussong, E. Lange, N. Pfeifer, O. Schulz-Trieglaff, A. Zerck, K. Reinert, and O. Kohlbacher. OpenMS: An open-source software framework for mass spectrometry. *BMC Bioinformatics*, 9, 2008.
62. Teradata Corporation. Teradata Temporal, Oct. 2010. `http://www.teradata.com/t/database/Teradata-Temporal/`, viewed October 2010.

DOI: 10.1002/spe

63. S. W. Thomas. The implementation and evaluation of temporal representations in XML. Master's thesis, Computer Science Department, University of Arizona, March 2009.
64. S. W. Thomas, 2011. http://research.cs.queensu.ca/~sthomas/tBench.html.
65. S. W. Thomas. Mining software repositories with topic models. Technical Report 2012-586, School of Computing, Queen's University, 2012.
66. S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein. Validating the use of topic models for software evolution. In *Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation*, pages 55–64, 2010.
67. S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein. Modeling the evolution of topics in source code histories. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 173–182, 2011.
68. S. W. Thomas, R. T. Snodgrass, and R. Zhang. τBench: Extending XBench with time. Technical Report TR-93, TimeCenter, 2010.
69. TPC. TPC Transaction Processing Performance Council. http://www.tpc.org/tpcc/, viewed July 14, 2011.
70. TREC. Text REtrieval Conference. http://trec.nist.gov/, viewed July 14, 2011.
71. E. Van der Vlist. *Relax Ng*. O'Reilly Media, 2004.
72. W3C. Document object model, 2007. http://www.w3.org/DOM, viewed March 26, 2007.
73. W3C. XQuery 1.0: An XML Query Language, W3C Recommendation, 2007. URL http://www.w3.org/TR/xquery, viewed November 5, 2011.
74. K. Walrath, M. Campione, A. Huml, and S. Zakhour. *The JFC Swing tutorial: A guide to constructing GUIs*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004.
75. Y. Wang, D. DeWitt, and J. Cai. X-Diff: An effective change detection algorithm for XML documents. In *Proceedings of the 19th International Conference on Data Engineering*, pages 519–530, 2004.
76. B. Yao, M. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *Proceedings of the 20th International Conference on Data Engineering*, pages 621–632, 2004.

DOI: 10.1002/spe