

A comparison of two approaches to utilizing XML in parametric databases for temporal data[☆]

Seo-Young Noh ^{*}, Shashi K. Gadia

Department of Computer Science, Iowa State University, Ames, IA 50010, USA

Received 12 April 2005; received in revised form 8 September 2005; accepted 9 October 2005

Available online 20 December 2005

Abstract

The parametric data model captures an object in terms of a single tuple. This feature eliminates unnecessary self-join operations to combine tuples scattered in a temporal relation. Despite this advantage, this model is relatively difficult to implement on top of relational databases because the sizes of attributes are unfixed. Since data boundaries are not problematic in XML, XML can be an elegant solution to implement parametric databases for temporal data. There are two approaches to implementing parametric databases using XML: (1) a native XML database with XQuery engine, and (2) an XML storage with a temporal query language. To determine which approach is appropriate in parametric databases, we consider four questions: the effectiveness of XML in modeling temporal data, the applicability of XML query languages, the user-friendliness of the query languages, and system performances of two approaches. By evaluating the four questions, we show that the latter approach is more appropriate to utilizing XML in parametric databases.

© 2005 Elsevier B.V. All rights reserved.

Keywords: Parametric database; Temporal database; Database comparison; XML utilization

1. Introduction

The parametric data model handles multi-dimensional data including temporal, spatial, spatiotemporal, and belief data. In this paper, we consider only time dimension. Therefore, the parametric data model can be considered as a temporal data model.

The parametric data model captures an object in terms of a single tuple, which is the main difference from other temporal data models that capture an object in terms of multiple tuples. To model an object in a single tuple, the parametric data model defines an attribute as a function of time. This modeling approach reduces query complexities, avoiding unnecessary self-join operations for combining tuples.

Despite the advantage, it is difficult to implement parametric data model on top of relational databases, the most popular approach to implementing temporal database systems. Since an attribute is a function of time, the size of an attribute value is unfixed. For example, John's salary has increased by \$1000

every year while Tom's salary has remained same. To capture the salary information, the size of two salary attributes should be different.

To resolve this problem, we need a flexible data description mechanism. Since data boundaries are not problematic in XML, XML can be an elegant solution for the parametric data model. XML and databases seem like an odd couple because they represent two different concepts driven by two different communities with different expectations and requirements [1]. Despite the differences, many researchers utilize XML in database research, leading to a new database generation.

When utilizing XML in parametric database implementation, we can consider two approaches:¹

- (1) Native XML database (NXD) with XQuery [3] engine.
- (2) XML storage with a temporal query language (ParaSQL).

To determine which approach is more appropriate in temporal databases, we conduct the following tasks:

[☆] This work has been sponsored in part by Information Infrastructure Institute and a grant from Baker Foundation Trust at Iowa State University.

^{*} Corresponding author.

E-mail address: rsyoung@cs.iastate.edu (S.-Y. Noh).

¹ In this paper, we do not consider an approach that uses an XML enabled relational database. This approach includes well-known problems such as *storing*, *retrieving*, and *round-tripping* of XML [2]. Therefore, considering this approach is beyond the scope of this paper.

- (1) Mapping the parametric temporal relation to an XML representation.
- (2) Introducing XQuery and ParaSQL (a temporal query language) queries for five plain English queries.
- (3) Comparing XQuery and ParaSQL queries with respect to user-friendliness.
- (4) Measuring system performances for two approaches.

The first task ascertains the usability of XML in temporal databases. The second task checks if XML query languages can be used for temporal databases.² The third task determines if XQuery provides user-friendliness for time-aspect queries. The last task compares performances of systems built by two approaches.

There are many native XML database systems, such as Software AG's Tamino [5], Apache Software Foundation's Xindice [6], and Wolfgang's eXist [7]. In this paper, we consider two open source native XML databases—Xindice and eXist.³ In the evaluation, we will compare these native XML databases (the first approach) with ParaDB (XML-based Parametric Database, the second approach).

Throughout the tasks, we will draw a conclusion that the second approach is more appropriate to utilizing XML in parametric databases for temporal data based on the following observations:

- (1) XML provides an elegant mechanism to describe the parametric data model without difficulties.
- (2) XQuery is possible for querying XML-based temporal databases. However, it is more complex than ParaSQL.
- (3) The target native XML databases are inadequate for large data sizes (1 GB) in processing temporal data while ParaDB achieves comparable performance and responds to an 1 GB database. ParaDB shows more favorable response times for queries where time processing is required.

The organization of this paper is as follows. In Section 2, we will overview temporal data models and XML in database research. In Section 3, we will provide a general introduction on the parametric data model for temporal data, and represent the data model in XML. Section 4 introduces XQuery and ParaSQL. Section 5 compares XQuery with ParaSQL for five English queries. Section 6 shows the implementation of the second approach. In Sections 7 and 8, we will discuss the experimental results and observations, respectively.

2. Related work

There are many different types of temporal data models and they have their own merits in their specific application areas. According to Jansen and Snodgrass's survey [8] in 1999, there are more than 2000 research papers on temporal databases.

Temporal data models can be categorized based on two criteria, such as *domain representation* and *timestamping schemes*. Each criterion consists of different modeling methods.

Domain representation indicates how a model represents a time domain of a value. There are three different domain representation methods: *point-based*, *interval-based*, and *temporal element-based*. Point-based time reference views time domain as a discrete, countable, infinite, linearly ordered set without end points. The individual element of the set represents the actual time instants while the linear order represents the progression of time. Examples of this data model are shown in SQL/TP [9] and SQL^T [10]. Interval-based scheme represents a domain of an object as the continuous maximum time interval. Interval-based temporal data models are introduced in TSQL2 [11] and IXQL [12]. Temporal element-based scheme represents domains of an object as finite unions of time intervals. Examples of this model are introduced in ParaSQL [13] and Tansel's NTC (Nested Relational Tuple Calculus) [14].

The timestamping determines whether a valid domain of a value is shared by all attributes in the same tuple. There are two timestamping methods: *tuple-level timestamping* and *attribute-level timestamping*.⁴ Tuple-level timestamping assigns a time domain to a single tuple so that all attributes share the same domain. Attribute-level timestamping, however, assigns time domains at attribute values.

Our proposed temporal database systems, NXDs and ParaDB, are based on the parametric data model that uses *temporal element-based* and *attribute-level timestamping* schemes.

Implemented temporal database systems are broadly categorized into two groups: temporal database systems over *relational database systems* and temporal database systems over *object-oriented database systems*. TIM [16] and TENORS [10] are examples for the former approach and TOOBIS [17] is an example of the latter approach.

In addition to conventional approaches to implementing temporal databases, there is an emerging trend that researchers are combining XML technology in database research. For example, SilkRoute [18] and XPERANTO [19] are middleware architecture between XML and relational database systems. They provide mechanisms to publish XML data from relational tables or store XML data into relational tables. TeXOR [20] is a temporal XML database working on an object-oriented relational database system.

As XML grows rapidly and provides flexible mechanisms to store and retrieve semistructured data, many commercial

² XPath [4] is another method to extract information from XML documents. XPath specifies the part of an XML document while XQuery provides a more sophisticated querying mechanism through FLWR expressions (For, Let, Where, and Return clauses). XQuery uses XPath in FLWR expression. Therefore, in this paper, we only compare XQuery with ParaSQL.

³ As of writing, Tamino allows maximum 20 MB XML file for evaluation so that we exclude it from our performance evaluations. Another reason for choosing two open source projects is that Xindice uses XPath for its query language while eXist uses XQuery.

⁴ Clifford et al. [15] classified it into two categories such as *temporally ungrouped* and *temporally grouped*.

companies and open source projects on NXDs have been launched. According to Bourret [21], there are more than 35 NXD systems. Software AG's Tamino is the representative commercial NXD, while Xindice and eXist are popular open source NXDs.

Wang and Zaniolo [22] provided a valuable insight on the usability of XML in temporal databases. They introduced an XML-based bitemporal data model and showed how XML could be used in temporal databases. In their another paper [23], they transformed time-aspect English queries into XQuery and provided performance evaluations between a native XML database and DB2.

3. Parametric data model for temporal data

In this section, the general concept of the parametric data model will be discussed. In the parametric data model, temporal elements are used to represent domains of objects. An attribute is defined as a function of time. Unlike most temporal data models, attribute values contain their own domains with values.

Fig. 1 shows temporal parametric relations. *Emp* and *Dept* relations maintain the history of employees and departments, respectively. Based on these relations, we will discuss the concept of temporal elements, temporal attribute values, tuples, and relations.

3.1. Temporal elements

Time intervals are inadequate to model the history of an object in a single tuple, and they lead to query languages that are difficult to express natural language queries [13]. Users frequently contain *or*, *and*, or (and) *not* conditions in their queries. These conditions are mapped to *union*, *intersection*,

and *complementation* operations in relational temporal data models. Since temporal databases handle time dimension, timestamps should be closed under these operations in order to obtain meaningful information after these operations.

To obtain timestamps that are closed under the set theoretic operations of union, intersection and complementation, the concept of *temporal elements* is introduced in the parametric data model [24–26]. The parametric data model assumes that there is a universe of time that consists of an interval $[0, \text{NOW}]$ of instants with a linear order $<$ on it. Here, NOW denotes the current instant of time. For simplicity, it is assumed that $[0, \text{NOW}]$ is the discrete set $\{0, 1, \dots, \text{NOW}\}$. A temporal element is a finite union of time intervals. A time interval is a temporal element. An instant t (or snapshot) may be identified with the interval $[t, t]$; thus, it is regarded as a temporal element. In *Emp* relation shown in Fig. 1(a), examples of temporal elements are $[11, 60]$ or $[0, 20] \cup [41, 51]$. The set of all temporal elements is closed under $\cup$, $\cap$, and $\neg$ (complementation with respect to $[0, \text{NOW}]$).

3.2. Temporal attribute values, tuples, and relations

To capture the changing value of an attribute, a *temporal value* of an attribute A is defined as a function of a temporal element into the domain of A . An example of a temporal value of *MName* attribute of the first tuple in *Dept* relation is $\langle [0, 50] \text{ Kim}, [71, \text{NOW}] \text{ Lee} \rangle$. If ξ is a temporal value, $\llbracket \xi \rrbracket$ denotes its domain. Thus $\llbracket \langle [0, 50] \text{ Kim}, [71, \text{NOW}] \text{ Lee} \rangle \rrbracket = [0, 50] \cup [71, \text{NOW}]$.

A tuple is a concatenation of temporal values. Informally, a temporal relation is defined as a set of temporal tuples.

3.3. XML representation

In a parametric temporal relation, each tuple has its own independent size because of unfixed sizes of attributes. This feature makes it difficult to adapt the parametric data model within relational databases. XML can be an elegant solution. Since XML does not restrict data boundaries, the parametric data model is simply transformed in an XML schema.

Fig. 2 shows the XML representations of *Emp* and *Dept* relations. Element $\langle \text{pdom} \rangle$ represents a parametric element (or temporal element). It may contain several $\langle \text{dunit} \rangle$ elements that represent time intervals. Each $\langle \text{value} \rangle$ node has its own domain. The union of all domains of $\langle \text{value} \rangle$ elements is the domain of an attribute. It is worth noting that each $\langle \text{tuple} \rangle$, $\langle \text{attribute} \rangle$, and $\langle \text{value} \rangle$ element has its own $\langle \text{pdom} \rangle$ element. It is designed to provide rapid responses to domain specific queries. This approach can retrieve domains of objects without stepping through large amount or entire data [27].

The two approaches compared in this paper use the same XML document based on an XML schema generated from the parametric data model. However, as we will discuss, it is worth noting that three systems transform the XML document in different forms internally. For example, Xindice compresses

<u>Name</u>	Salary	DName
[11,60] John	[11,49] 50K	[11,44] R&D
	[50,54] 55K	[45,60] Test
	[55,60] 60K	
[0,20] Tom $\cup$ [41,51]	[0,20] 45K	[0,20] Sales
	[41, 51] 50K	$\cup$ [41,51]

(a) *Emp* relation

<u>DName</u>	MName
[0,50] R&D $\cup$ [71,NOW]	[0,50] Kim
	[71,NOW] Lee
[11,NOW] Test	[11,44] Leu
	[45,NOW] Inga
[10,NOW] Sales	[10,NOW] Smith

(b) *Dept* relation

Fig. 1. A parametric temporal database.

```

<Relation name="Emp">
  <tuple>
    <attribute name="Name">
      <value>John
      <pdom><dunit start="11" end="60"/>
    </pdom>
    </value>
    </attribute>
    <attribute name="Salary">
      <value>50K
      <pdom><dunit start="11" end="49"/>
    </pdom>
    </value>
    <value>55K
    <pdom><dunit start="50" end="54"/>
    </pdom>
    </value>
    <value>60K
    <pdom><dunit start="55" end="60"/>
    </pdom>
    </value>
    </attribute>
    <attribute name="DName">
      <value>R&D
      <pdom><dunit start="11" end="44"/>
    </pdom>
    </value>
    <value>Test
    <pdom><dunit start="45" end="60"/>
    </pdom>
    </value>
    </attribute>
    <pdom><dunit start="11" end="60"/>
  </pdom>
</tuple>
<!-- the other tuple is omitted -->
</Relation>

```

(a) emp.xml

```

<Relation name="Dept">
  <tuple>
    <attribute name="DName">
      <value>R&D
      <pdom>
        <dunit start="0" end="50"/>
        <dunit start="71" end="NOW"/>
      </pdom>
    </value>
    </attribute>
    <attribute name="MName">
      <value>Kim
      <pdom>
        <dunit start="0" end="50"/>
      </pdom>
    </value>
    <value>Lee
    <pdom>
      <dunit start="71" end="NOW"/>
    </pdom>
    </value>
    </attribute>
    <pdom>
      <dunit start="0" end="50"/>
      <dunit start="71" end="NOW"/>
    </pdom>
  </tuple>
  <!-- the rest tuples are omitted -->
</Relation>

```

(b) dept.xml

Fig. 2. XML documents for Emp and Dept temporal relations.

the document while XML-based parametric database paginates it into smaller pages.

4. XQuery and ParaSQL

4.1. XML query language

In this section, we will briefly discuss XQuery. XQuery is derived from an XML query language called Quilt, developed by Chamberlin et al. [28]. Quilt borrowed some syntax and concepts from SQL, ODMG, XPath 1.0, XQL and XML-QL [3]. XQuery is a functional language in which a query is represented as an expression called FLWR expression. The acronym FLWR stands for the keywords FOR, LET, WHERE, and RETURN, which occur in FLWR expression. The following shows a simplified BNF of XQuery.

```

<flwrExpr> ::= (<forClause> | <letClause>)+
             <whereClause>? RETURN <expr>
<forClause> ::= FOR <variable> IN
             <expr> (, <variable> IN <expr>)*
<letClause> ::= LET <variable> :=
             <expr> (, <variable> := <expr>)*
<whereClause> ::= WHERE <expr>

```

FOR and LET can be used in any sequence and any number of times. Each FOR and LET can declare several variables. FOR creates an iteration while LET binds to a group of nodes, creating *no* iteration. WHERE filters out nodes that do not pass a condition, and RETURN constructs a result. WHERE and RETURN clauses are applied to each iteration. A FLWR expression can contain any sequence of FOR, LET, and WHERE clauses, provided at least one such clause is present, followed by a RETURN clause [3]. For an illustration, consider the following query:

Query: List manager's names who managed Software department throughout 10 to 30.

XQuery:

```

for $d in document("dept.xml")//tuple
let $te:=restriction($d/pdom, telement("10", "30"))
let $dname:= $d/attribute[@name="DName"]/value
let $mname:= $d/attribute[@name="MName"]/value
where $dname = "Software" and not (empty($te))
return $mname/text( )

```

In this XQuery example, we use functions such as restriction, telement and empty. These functions are predefined by users before executing the query.

The variable \$d iterates all tuples in dept.xml. The restriction function restricts the domain of a tuple to time interval [10,30]. Variable \$te contains a restricted domain (or simply temporal element). In WHERE clause, it is checked that a department name is Software and \$te is not empty. The RETURN clause returns manager name(s).

4.2. Parametric structured query language

ParaSQL consists of three expressions: *relational expression*, *domain expression*, and *boolean expression*. They evaluate relations, temporal elements, and boolean values, respectively. These three expressions are mutually recursive.

4.2.1. Relational expression

A relational expression returns a relation that is a set of temporal tuples. It can be expressed by UNION, INTERSECTION, and SELECT statement. In this discussion, we only concentrate on SELECT statement because it is the most interesting relational expression. SELECT statement in ParaSQL is SQL-style SELECT statement. The following shows the BNF of the SELECT statement.

```
<relational expression> ::=
    SELECT <attribute list>
    [RESTRICTED TO <domain expression>]
    FROM <relation list>
    [WHERE <boolean expression>]
```

Unlike classical SQL, it has RESTRICTED TO clause that restricts the domain of tuples qualified by WHERE clause. WHERE clause in ParaSQL has the same functionality of SQL, that is, it returns a tuple if a boolean expression is satisfied.

4.2.2. Domain expression

A domain expression is used to restrict the domain of tuples (or objects) filtered by a boolean expression. The following shows the BNF of domain expression.

```
<domain expression> ::=
    [[<attribute>]] |
    [[<attribute>  $\theta$  <attribute>]] |
    [[<attribute>  $\theta$  <value>]] |
    [[<relational expression>]] |
    <temporal element>
```

The domain expression, [[<attribute>]], collects temporal domain of a specified attribute. The domain expressions, [[<attribute> θ <attribute>]] and [[<attribute> θ <value>]], collect temporal domains such that the θ is satisfied, where θ is an arithmetic operator like > and $\neq$. The domain expression, [[<relational expression>]], collects all temporal domains of tuples returned by a relational expression. The last type of domain expression is temporal element, which can be a time instant. Domain expressions can be connected by $\cup$, $\cap$, and $\neg$ (complementation).

4.2.3. Boolean expression

A boolean expression determines if a given tuple satisfies boolean conditions. A boolean expression has the same functionality as classical SQL in that it either qualifies or disqualifies a tuple. But it differs from classical SQL in that it can be constructed by domain expressions with set operations. For example, suppose a user wants to retrieve Software department's information. In WHERE clause, it can be expressed like "DName=Software." However, this expression is abbreviation of [[DName='Software']] $\neq \emptyset$, meaning that if the domain of DName whose value is Software is not empty, then the tuple has information about Software department. The following shows the BNF of boolean expression.

```
<boolean expression> ::=
    <domain expression> set op <domain expression> |
    <attribute>  $\theta$  <attribute> |
    <attribute>  $\theta$  <value>
```

For an illustration of ParaSQL, revisit the query that we expressed in XQuery.

Query: List manager's names who managed Software department during [10, 30].

ParaSQL:

```
SELECT D.MName
RESTRICTED TO [10,30]
FROM Dept D
WHERE D.DName = 'Software'
```

The ParaSQL query retrieves tuples from Dept relation and evaluates them if the department name is Software. Qualified tuples by the WHERE clause are restricted to interval [10,30]. Therefore, if the intersection of [10,30] and the domain of a qualified tuple is not empty, the manager name of the selected tuple is returned.

5. Query comparison

Wang and Zaniolo [23] introduced English queries categorized into five types such as *relation scan*, *history*, *interval*, *snapshot*, and *temporal join*. They expressed five English queries into XQuery and showed XQuery can be used in temporal databases. Their XQuery queries are slightly different from ours because data models are different (interval-based vs. temporal element-based). We will compare XQuery with ParaSQL for five English queries.⁵ The XQuery queries discussed in this section are based on emp.xml and dept.xml XML documents shown in Fig. 2. The ParaSQL queries are based on Emp and Dept relations shown in Fig. 1.

• **Query 1:** Retrieve all employee information. (Relation scan)

XQuery:

```
for $e in document("emp.xml")//tuple
return $e
```

⁵ XPath is excluded from our discussion because XQuery is including XPath. However, we will evaluate systems for possible XPath queries in Section 7.

ParaSQL:

```
SELECT *
FROM Emp E
```

The XQuery query iterates tuples indicated by variable \$e. Each <tuple> element is returned by RETURN clause.

The ParaSQL query returns all information from Emp relation. Since there are no domain restriction and boolean conditions, it does not have RESTRICTED TO and WHERE clauses. However, the omitted clauses are recovered in an expression tree. Therefore, the ParaSQL query can be expressed as follows:

```
SELECT *
RESTRICTED TO [0, NOW]
FROM Emp E
WHERE TRUE
```

- **Query 2:** Retrieve the salary history of employee John. (History)

XQuery:

```
for $e in document("emp.xml")//tuple
let $sal := $e/attribute[@name = "Salary"]
where $e/attribute[@name = "Name"]/value = "John"
return $sal
```

ParaSQL:

```
SELECT E.Salary
FROM Emp E
WHERE E.Name = 'John'
```

The XQuery query iterates <tuple> element pointed by \$e variable. The variable \$sal points an <attribute> element whose attribute name is Salary. By the WHERE clause, the <tuple> element is evaluated if the name of employee is John. Finally it returns John's salary information.

The ParaSQL is straightforward. The query retrieves employee John's information only. However, it must be noted that it is only true when Name attribute is the key attribute of Emp relation. Since John is a unique name in the relation, just retrieving John's tuple is enough for this query. If Name attribute is *not* a key attribute, we must restrict the domain of the tuple to John's domain. Therefore, the ParaSQL can be expressed as follows:

```
SELECT E.Salary
RESTRICTED TO [[E.Name = 'John']]
FROM Emp E
```

In another version of ParaSQL, RESTRICTED TO clause is used to select John's information. The domain expression, [[E.Name='John']], says that we are interested in the domain of employee whose name is John. Therefore, a tuple is restricted to John's domain, leading to eliminating the other information (John's information is a part of history of the tuple).

- **Query 3:** Find all the employee names throughout time instant 20–50. (Interval)

XQuery:

```
for $e in document("emp.xml")//tuple
let $name := $e/attribute[@name = "Name"]/value
let $te := restriction($e/pdom, telement("20","50"))
where not (empty($te))
return $name/text()
```

ParaSQL:

```
SELECT E.Name
RESTRICTED TO [20,50]
FROM Emp E
```

In the XQuery query, variable \$name points an element that contains an employee's name. By using restriction and telement predefined functions, variable \$te contains an element that represents the intersection of a temporal element of an employee tuple and time interval [20,50]. In WHERE clause, the intersection is evaluated if \$te is empty, meaning there is no common temporal element. If it is not empty, then the query returns the employee's name.

In ParaSQL query, RESTRICTED TO clause indicates time of interests. Therefore, all employee's domains are restricted to time interval [20,50]. If there is no common time instants between the domain of employee and interval [20,50], the employee is eliminated.

- **Query 4:** Find all employee names who worked at time instant 17. (Snapshot)

XQuery:

```
for $e in document("emp.xml")//tuple
let $name := $e/attribute[@name = "Name"]/value
let $te := restriction($e/pdom, telement("17","17"))
where not (empty($te))
return $name/text()
```

ParaSQL:

```
SELECT E.Name
RESTRICTED TO [17,17]
FROM Emp E
```

The XQuery and ParaSQL queries are almost identical to Query 3. Since snapshot is another type of interval, two queries are similarly expressed to their interval queries.

In Queries 3 and 4, we only retrieve names of employees. These types of queries are simply expressed in XQuery with hypothetical functions, restriction and telement. However, if the queries are changed to retrieve entire information about employees during an interval or a time instant, XQuery queries require an additional processing mechanism. For example, consider the following query:

Find all employee information who worked at time instant 17 (or time interval [20,50]).

Because qualified tuple's domain should be restricted to a specific domain, reconstructing an XML document is

unavoidable. It may be claimed that just adding a specific domain with tuple information is enough. However, some part of employee information may be invalidated in the specific time domain. Therefore, the invalidated information should be eliminated from the employee's information. We must note that the invalidated information is a part of employee information. Unlike XQuery, ParaSQL does not require such additional processing because only tuple's validated information is passed by RESTRICTED TO clause.

• **Query 5:** List all pairs of department and employee names such that the employee worked or is working in the department. (Temporal join)

XQuery:

```
for $d in document("dept.xml")/tuple
let $dname := $d/attribute[@name="DName"]/value
let $dpdom := $d/pdom
for $e in document("emp.xml")/tuple
let $ename := $e/attribute[@name="Name"]/value
let $edname := $e/attribute[@name="DName"]/value
let $epdom := $e/pdom
where $dname = $edname and
not (empty(intersection($dpdom, $epdom))
return
<list>
  $ddname/text()
  $edname/text()
</list>
```

ParaSQL:

```
SELECT D.Dname, E.Name
RESTRICTED TO [[D.Dname=E.Dept]]
FROM Dept D, Emp E
```

In the XQuery query, two department names from `emp.xml` and `dept.xml` are compared. Each department's domain is intersected with each employee's domain. If two department names are the same and the intersection is not empty, there exists information such that the employee worked or is working in the department. In RETURN clause, the XQuery query creates an element containing a department name and an employee name.

In the ParaSQL query, the temporal join is expressed with RESTRICTED TO clause. If θ comparison ($=$) is satisfied, combined tuples are restricted to the domain such that the θ condition is satisfied. Therefore, all attributes in the same tuple have the same temporal domain.

By expressing five English queries into XQuery and ParaSQL queries, we can note that XQuery is relatively more complex than ParaSQL. Since XQuery is not designed for temporal databases, it requires additional predefined functions leading to complex expressions. We have also discussed that if a query asks entire information about an object restricted by a specific domain, an expensive computational element processing is required because some parts of information, outside of the specific domain, should be eliminated from the original information.

Despite the advantages of XML in modeling complex temporal data, XQuery is inadequate for querying temporal

databases because of the lack of user-friendless in temporal queries.

6. ParaDB implementation

In this section, we will discuss the parametric database system (ParaDB). We implemented ParaDB based on the second approach, an XML storage with a temporal query language (ParaSQL). ParaDB is an XML-based system and uses a three-layer system architecture which consists of Query Processing Layer (Layer-1), Query Execution Layer (Layer-2), and Storage Management Layer (Layer-3) [29,30].

Layer-1 focuses on query processing such as parsing ParaSQL queries, generating expression trees and query plans.⁶ Layer-2 executes the ParaSQL queries using DOM (Document Object Model) [31] API. The parse trees and expression trees are all XML documents. Layer-3 is a paginated XML storage. This storage provides customized XML DOM API to access and retrieve temporal tuples stored in paginated XML document.

6.1. Query Processing Layer

A ParaSQL parser parses ParaSQL queries and passes the parse tree to a logical query planner. By the logical query planner, a parse tree is transformed into an expression tree. Fig. 3(a) shows the expression tree for Query 5 discussed in Section 5. The expression tree is an abstract-level description for the query, and it is expressed in XML as shown in Fig. 3(b). In the XML representation, `<projection>`, `<restriction>`, and `<where>` elements are at the same level that is different from the abstract expression tree. However, they have the same functionality because two representations are traversed by depth first search. The XML representation has annotation nodes to provide information for a query executor. For example, an annotation for a join can indicate which iterator should be used for query execution. We must note that the expression tree is a *standard form*, not an optimized form. It is up to the optimizer to adjust the expression tree.

Representing parse trees and expression trees in XML is of great benefit to the implementation of ParaDB. XML can help to reduce implementation complexities because our reliance on linked-list data structure can be replaced by the DOM navigation mechanism, leading to more human readable and reliable codes.

6.2. Query Execution Layer

The Query Execution Layer executes a ParaSQL query represented in an expression tree. A query executor in this layer uses DOM API to understand an expression tree and to execute it. Algorithm 1 shows the query execution procedure.

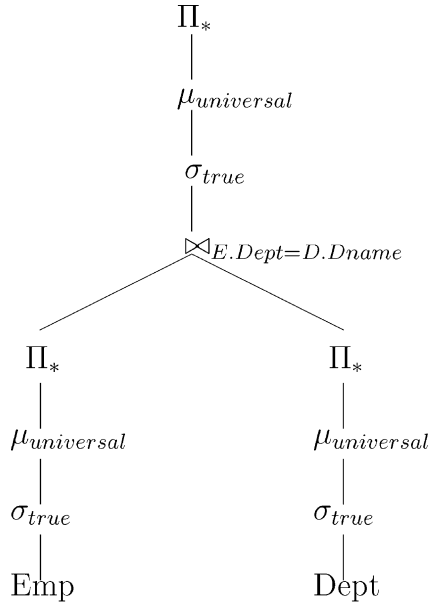
⁶ In this discussion, we do not consider optimizer and query planner. Therefore, an expression tree is directly passed to a query executor.

Algorithm 1 Query Execution Algorithm

```

1: procedure QUERYEXECUTION( $e$ )                                ▷  $e$ : exp. tree
2:   if  $e$  has join condition then
3:      $it \leftarrow \text{JOIN}(e)$                                     ▷  $it$ : iterator
4:   else if  $e$  has a relation scan then
5:      $it \leftarrow \text{RELATIONSCAN}(e)$ 
6:   end if
7:   while  $it.\text{hasNext}() = \text{true}$  do
8:      $\text{tuple} \leftarrow it.\text{getNext}()$                             ▷ retrieve a tuple
9:      $\text{tuple} \leftarrow \text{RESTRICTION}(e, \text{tuple})$ 
10:    if  $\text{tuple} \neq \text{null}$  then
11:       $\text{output}(\text{tuple})$                                        ▷ write a tuple
12:    end if
13:  end while
14: end procedure

```



(a) Expression tree of Query 5

```

<ExpressionTree>
  <Projection>
    <AttributeList>
      <Attribute relName="*" attrName="*" />
    </AttributeList>
  </Projection>
  <Restriction>
    <DomainExpression type="universal" />
  </Restriction>
  <WhereCondition>
    <BooleanExpression type="true">
      <Iterator>
        <Annotation iterator="NestedLoopJoin" />
        <JoinCondition>
          <!--Annotations for join condition-->
        </JoinCondition>
        <OuterRelation>
          <!--Outer relation information-->
        </OuterRelation>
        <InnerRelation>
          <!--Inner relation information-->
        </InnerRelation>
      </Iterator>
    </BooleanExpression>
  </WhereCondition>
</ExpressionTree>

```

(b) XML representation

Fig. 3. Expression tree and its XML representation.

It determines which iterator should be used to process a given expression tree. The expression tree has information about an iterator to be used. Once an iterator is determined, the query executor retrieves tuples from the storage by using the iterator, and the iterator qualifies tuples using EVALUATION function. The qualified tuples are restricted to a temporal domain by RESTRICTION function.

The most important functions used to execute queries are EVALUATION and RESTRICTION functions for WHERE and RESTRICTED TO clauses, respectively. Even though they are different functionalities, they share some base functions because they are mutually recursive.

To clarify how ParaSQL queries are evaluated, consider a domain expression $\llbracket A\theta B \rrbracket$. This domain expression returns

```

<DomainOp opType = "unary">
  <BinaryOp opType = "=">
    <Attribute relName="Dept" attrName="DName" type="string" attrPos="0" />
    <Attribute relName="Emp" attrName="DName" type="string" attrPos="2" />
  </BinaryOp>
</DomainOp>

```

Fig. 4. XML representation of $\llbracket [D.DName = E.DName] \rrbracket$.

a domain such that two attribute values, A and B , have the θ relationship. For example, Query 5 discussed in Section 5 uses a domain expression $[[D.DName=E.DName]]$ in the RESTRICTED TO clause. Fig. 4 shows the domain expression represented in XML.

operation. If the operation satisfies, the common domain (or intersected domain) of two nodes will be returned.

The other operations such as BOOLEANEXPRESSION and RELATIONALEXPRESSION can be explained in the same way as DOMAINEXPRESSION because they are mutually recursive.

Algorithm 2 Query Evaluation Algorithm

```

1: procedure NEXTEVAL( $e, tup1, tup2$ )     $\triangleright e$ : expression node,  $tup$ : tuple
2:   if  $e$  is a type of boolean expression then
3:      $result \leftarrow$  BOOLEANEXPRESSION( $e.getFirstChild(), tup1, tup2$ )
4:   else if  $e$  is a type of domain expression then
5:      $result \leftarrow$  DOMAINEXPRESSION( $e.getFirstChild(), tup1, tup2$ )
6:   else if  $e$  is a type of relational expression then
7:      $result \leftarrow$  RELATIONALEXPRESSION( $e.getFirstChild(), tup1, tup2$ )
8:   end if
9:   return  $result$                                  $\triangleright result$  is an abstract type
10: end procedure

11: procedure DOMAINEXPRESSION( $e, tup1, tup2$ )
12:   if  $e$  is a binary operation then
13:      $u \leftarrow$  getFirstOperand( $e$ )
14:      $v \leftarrow$  getSecondOperand( $e$ )
15:      $op \leftarrow$  getOperation( $e$ )
16:      $a \leftarrow$  getNode( $u, tup1$ );     $\triangleright$  get a node from  $tup1$  for the operand
17:      $b \leftarrow$  getNode( $v, tup1$ )
18:     while  $a \neq null$  do
19:       while  $b \neq null$  do
20:          $i \leftarrow$  getValue( $a$ )       $\triangleright$  get a value from node  $a$ 
21:          $j \leftarrow$  getValue( $b$ )
22:         if  $(i \text{ op } j) = true$  then     $\triangleright$  evaluate  $i$  and  $j$  on  $op$ 
23:            $domain \leftarrow$  intersectDom( $a, b$ )   $\triangleright$  find common domain
24:         end if
25:          $b \leftarrow b.getNextSibling$ 
26:       end while
27:        $b \leftarrow$  getNode( $v, tup1$ )
28:        $a \leftarrow a.getNextSibling()$ 
29:     end while
30:   else if  $e$  is a unary operation then
31:     ...
32:   end if
33:   return  $domain$ 
34: end procedure

```

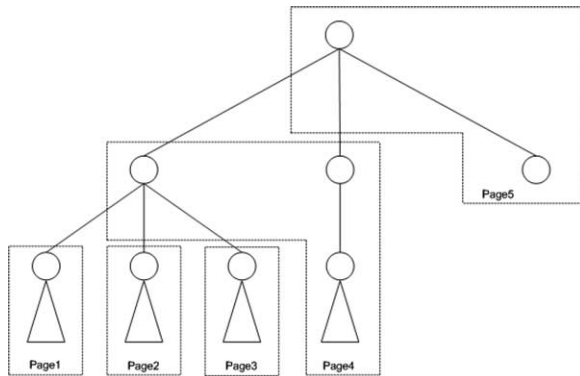
The element `<DomainOp>` is the root node and is passed to function NEXTEVAL as an argument with tuples. NEXTEVAL function determines which function should be called to process the current expression node. In our example, DOMAINEXPRESSION function is called. It is worth noting that in NEXTEVAL function there are three different functions used to process boolean expression, domain expression, relational expression. Since these three expressions are mutually recursive, they can call NEXTEVAL function recursively. Algorithms 2 shows query evaluation procedures.

In our example, there is a binary operation in the domain expression. From the expression tree, the algorithm extracts two operands and one operation. Since ParaDB is XML-based, the algorithm retrieves two nodes for the operands. It iterates values from the nodes and evaluates them by using the

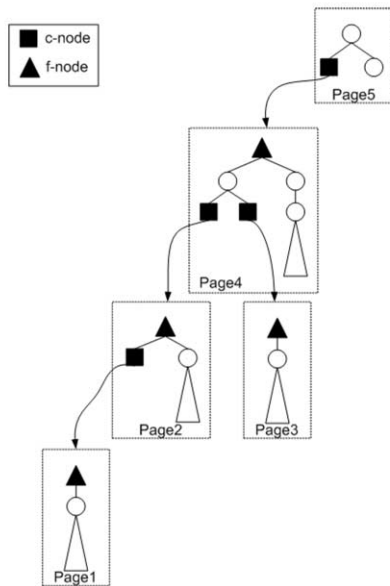
6.3. Storage Management Layer

The Storage Management Layer handles page requests from its upper layer. It provides a requested page from a disk. This layer manages paginated XML data. Whenever it receives a request, it retrieves one page at a time from the disk. It provides DOM API so that iterators can retrieve nodes from loaded pages. This layer has a buffer manager to reduce the number of disk accesses for repeatedly used pages.

Kanne and Moerkotte [32] invented the Natix storage that paginates a large XML document into a set of pages. Natix is a storage technology for XML documents. We have developed our own storage technology for XML called CanStoreX (Canonical Storage for XML) [33]. In order to facilitate pagination of an XML document, Natix as well as our storage



(a) An XML document



(b) Paginated XML document

Fig. 5. XML document and paginated XML document.

Table 1

Test data information

No.	Notation	XML size (KB)	DB size (KB)	Pages	Capacity	Employees
1	10 KB	12	20	5	0.59	3
2	100 KB	105	152	38	0.69	36
3	1 MB	1027	1472	368	0.70	363
4	10 MB	10,242	14,680	3670	0.70	3633
5	100 MB	102,420	146,784	36,696	0.70	36,330
6	1 GB	1,050,137	1,505,000	376,250	0.70	372,385

Table 2

Internal system configuration

Configuration	Xindice	eXist	ParaDB
Data	Compression	Paged	Paged
Index	Yes	Yes	No
Optimizer	Yes	Yes	No

7.1. Data and system configuration

There are many XML data synthesizers such as ToXgene [34] and IBM XML Generator [35]. However, we found that it is difficult to synthesize XML data that has time features. Therefore, the test data set is generated by our own definitions as follows:

- (1) Data sizes increase 10 times from approximately 10 KB to 1 GB.
- (2) Employee information increases only when increasing the size of databases.
- (3) Each employee tuple has salary and department history information for more than 30 years (from 1970 to 2004).
- (4) In employee tuples, salary increases \$100 every year and department information is updated every 5 years.

Table 1 shows the information about the XML test data.⁷ There are six different databases that are growing 10 times from 12K (≈ 10 KB) to 1,050,137K (≈ 1 GB).⁸

The capacity column of Table 1 shows the average occupation of pages in each database. The number of employee's column shows how many employee tuples reside in each database, where the approximate size of an employee tuple is 2800 bytes.⁹

Table 2 shows the internal configurations about open source native XML databases and ParaDB.

⁷ Xindice and eXist use the actual XML documents. DB size, pages, and capacity columns are only for ParaDB.

⁸ In the synthesized data generation, we tried to keep the XML data size to 10^i KB, where $i = 1, \dots, 6$. Each XML database is paginated into 4 KB pages for ParaDB.

⁹ In this paper, we do not consider a database such that an employee information resides in multiple pages. It will be considered in our future work.

technology adds some auxiliary nodes to the document. Whereas in Natix a page consists of several (small) XML elements, our pages are self-contained XML documents on their own right. To a client of our DOM API, auxiliary nodes and page boundaries are transparent.

Fig. 5 shows an XML document and a corresponding paginated XML document. In the paginated XML document, a *c-node* contains a page ID pointing to a child node which resides in another page while an *f-node* groups a sequence of one or more children nodes to which are pointed by a *c-node*. The detailed explanation on CanStoreX is beyond the scope of this paper.

7. Performance comparison

In this section, we will discuss system performances for two native XML databases (the first approach) and ParaDB (the second approach).

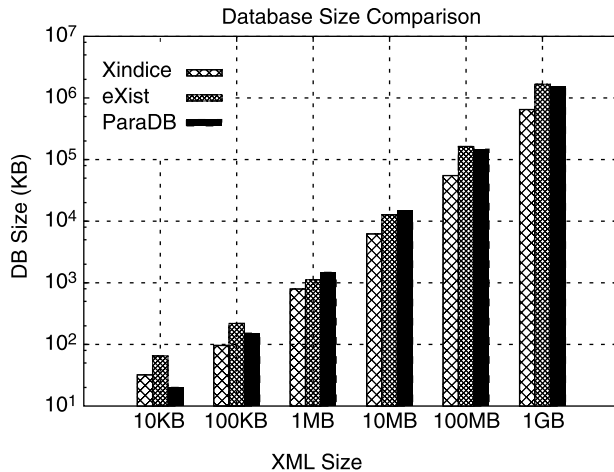


Fig. 6. Actual XML sizes and corresponding DB sizes.

7.2. Database size comparison

Fig. 6 shows the comparisons between actual XML data sizes and corresponding database sizes. Xindice compresses the XML data so that the database sizes are less than the actual XML data sizes if the data sizes are greater than 100 KB. Unlike Xindice, eXist and ParaDB increase the database sizes for the XML data because they paginate the XML documents into small pages.

7.3. Experimental results

Fig. 7 shows experimental results for the five queries.¹⁰ For Query 1, eXist shows the best performance while Xindice and ParaDB have the similar performance results. However, eXist and Xindice do not respond to the 1 GB database.

For Query 2, Xindice and ParaDB show a similar response time and responded to the 1 GB database. eXist has the rapid response time for some databases, but it does not respond to the 1 GB database.

For Query 3, Xindice and ParaDB provide faster responses than eXist. However, only ParaDB responds to the 100 MB and 1 GB databases. We must note that since XPath cannot modify the selected outputs, the XPath in this query has been tested without processing interval computations. Therefore, it can be assumed that the response time is the optimal time of Xindice.

For Query 4, the response times are similar to the experimental results for Query 3. As noted in Section 5, despite the fact that two queries are different types, they have similar properties. Xindice and ParaDB show the rapid

response time over eXist, but Xindice and eXist do not respond to the 1 GB database.

For Query 5, only eXist and ParaDB's response times are measured because XPath used by Xindice lacks the ability to express join operations. ParaDB shows better performance than eXist up to 1 MB while eXist shows better performance for 10 and 100 MB. However, only ParaDB responds to 1 GB.¹¹

As we have seen in the performance results, Xindice and eXist had difficulties in handling large XML data files like 1 GB. It is worth noting that Query 1, Query 2, and Query 5 are queries in which time processing is *not* required (scan, history, and join). For those queries, eXist provides faster responses than ParaDB. However, for the other queries where time processing is required, ParaDB provides faster response times than the other systems.

Even though ParaDB does not use any index and optimization, the experimental results show that ParaDB provides comparable performances to Xindice and eXist for the five queries.

8. Conclusion

The parametric data model captures an object in terms of a single tuple. This property removes unnecessary self-join operations, leading to reduction of query complexities. Despite this advantage, it is relatively difficult to implement on top of relational databases, which is the most popular approach in temporal database implementations. Since an attribute is defined as a function of time, the sizes of attribute values are unfixed. This difficulty, however, can be resolved by XML because in XML data boundary is not problematic. In Section 3, we have shown how the parametric data model could be represented in XML.

When adapting XML in parametric data model, we could consider two approaches such as native XML database with XQuery engine and XML storage technology with a temporal query language. In order to determine which approach is more appropriate in parametric databases, we have considered four questions: the applicability of XML in temporal databases, XQuery's expressibility for temporal queries, user-friendliness, and system performances.

In Section 4, we have compared ParaSQL with XQuery. For the five queries, we have seen that XQuery can express the queries. However, the expressions are more complex than ParaSQL's. We have also noted that if a query asks entire information about an object for a specific domain, an expensive element processing is expected in XQuery because some parts of information should be eliminated from the original information.

We have tested two native open source XML databases and the XML-based parametric database system. Throughout the performance evaluation, ParaDB has provided faster response

¹⁰ The test platform is a Pentium 4 2 GHz PC with 512 MB memory and 40 GB IDE hard drive under Windows XP operating system. For more accurate experimental results, measuring number of disk accesses are required. However, Xindice and eXist do not provide functions to analyze page request or access information.

¹¹ Even though eXist claims that XQuery engine passed 92% of use cases provided by W3C, it is still inadequate to process intervals. Therefore, we tested eXist without processing interval computations for outputs. However, we tested ParaDB without any restrictions.

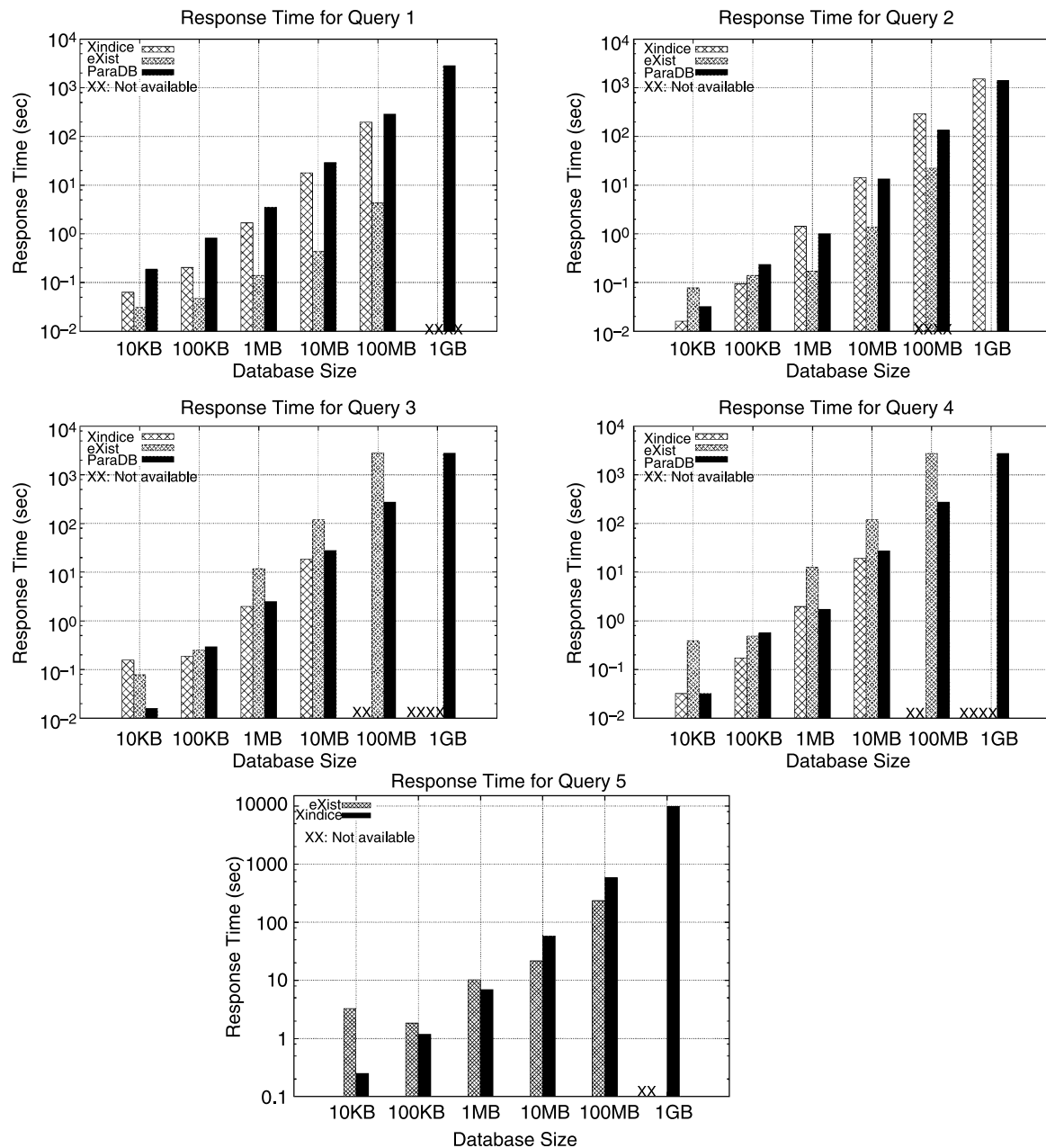


Fig. 7. Query response times for five queries.

times than Xindice and eXist for the queries where time processing is required. Even though Xindice and eXist have achieved better performance than ParaDB for relation scan and join queries, they do not respond to an 1 GB database. Since ParaDB does not use any indexes, ParaDB is relative slower than the other systems for history and join queries. Therefore, it is expected that ParaDB will provide a comparable performance to Xindice and eXist for those queries if indexes are available to use.

Throughout our tasks, we can summarize our observations as follows:

(1) XML provides an elegant mechanism to describe the parametric data model without difficulties.

- (2) XQuery is possible for querying XML-based temporal databases. However, it is more complex than ParaSQL.
- (3) The two native XML databases are inadequate for large data sizes (1 GB) in processing temporal data. ParaDB achieves comparable performance as well as responds to an 1 GB database. ParaDB shows more favorable response times for queries in which time processing is required.

Consequently, we can conclude that the second approach is more appropriate to utilizing XML in parametric databases for temporal data

XML is an emerging technology and is adapted in many research areas. Temporal databases are no exception. Since XML provides a flexible mechanism to represent complex

temporal data, XML can be an elegant option for implementations of temporal databases (or multi-dimensional databases). In this paper, we have shown how XML can be used in temporal databases. By comparing two approaches, we could determine which approach is more suitable to utilizing XML in parametric databases.

References

- [1] A. Schmidt, et al., Why and how to benchmark XML databases, ACM SIGMOD Record 30 (3).
- [2] R. Bourret, XML and databases, Website: <http://www.rpbouret.com/xml/XMLandDatabases.htm>, April 2005.
- [3] W3C, XML query, Website: <http://www.w3.org/XML/Query>, December 2004.
- [4] W3C, XML path language (XPath), Website: <http://www.w3.org/DOM>, April 2005.
- [5] Software AG, Tamino native XML server, Website: <http://www2.softwareag.com/Corporate/products/tamino>, December 2004.
- [6] Apache Foundation, Apache Xindice, Website: <http://xml.apache.org/xindice/>, December 2004.
- [7] M. Wolfgang, Open source native database (eXist), Website: <http://exist.sourceforge.net/>, December 2004.
- [8] C.S. Jensen, R.T. Snodgrass, Temporal database management, IEEE Transactions on Knowledge and Data Engineering 11 (1) (1999) 464–497.
- [9] D. Toman, Point-based temporal extension of temporal SQL, in: Proceedings of the Fifth International Conference on Deductive and Object-Oriented Databases, 1997, pp. 103–121.
- [10] C.X. Chen, J. Kong, C. Zaniolo, Design and implementation of a temporal extension of SQL, in: Proceedings of the 19th International Conference, on Data Engineering, Bangalore, India, 2003.
- [11] R.T. Snodgrass (Ed.), The TSQL2 Temporal Query Language, Kluwer, Dordrecht, 1995.
- [12] N.A. Lorentzos, Y.G. Mitsopoulos, SQL extension for interval data, IEEE Transactions Knowledge on Data Engineering 9 (3) (1997) 480–499.
- [13] S.K. Gadia, S.S. Nair, Temporal databases: a prelude to parametric data, in: Temporal Databases: Theory, Design, and Implementation, Benjamin/Cummings, 1993, pp. 28–66.
- [14] A.U. Tansel, E. Tin, The expressive power of temporal relational query languages, IEEE Transactions on Knowledge and Data Engineering 9 (1) (1997) 120–134.
- [15] J. Clifford, et al., On temporal grouping, in: Proceedings of the International Workshop on Temporal Databases, 1995, pp. 194–213.
- [16] J. Yang, H.C. Ying, J. Widom, TIP: a temporal extension to informix, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, 2000.
- [17] TOOBIS Esprit Project, Toobis: temporal object-oriented databases within information systems, Technical Report T12R.3, <http://www.mm.di.uoa.gr/toobis/>, June 2005 (accessed April 1, 2005).
- [18] M.F. Fernandez, W.C. Tan, D. Suciu, Silkroute: trading between relations and XML, Computer Networks 33 (1–6) (2000) 723–745.
- [19] M.J. Carey, et al., XPERANTO: middleware for publishing object-relational data as XML documents, in: Proceedings of 26th International Conference on Very Large Data Bases, 2000, pp. 646–648.
- [20] K. Nørvåg, M. Limstrand, L. Myklebust, TeXOR: temporal XML database on an object-relational database system, in: Ershov Memorial Conference, 2003, pp. 520–530.
- [21] R. Bourret, XML database products, Website: <http://www.rpbouret.com/xml/XMLDatabaseProds.htm>, December 2004.
- [22] F. Wang, C. Zaniolo, Xbit: an XML-based bitemporal data model, in: Proceedings of the 23rd International Conference on Conceptual Modeling, Shanghai, China, 2004.
- [23] F. Wang, C. Zaniolo, Publishing and querying the histories of archived relational databases in XML, in: Proceedings of the Fourth International Conference on Web Information Systems Engineering, Rome, Italy, 2003.
- [24] S.K. Gadia, J.H. Vaishnav, A query language for a homogeneous temporal database, in: Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Portland, OR, USA, 1985, pp. 51–56.
- [25] S.K. Gadia, A homogeneous relational model and query languages for temporal databases, ACM Transactions on Database Systems 13 (4) (1988) 418–448.
- [26] C.S. Jensen, et al., A glossary of temporal database concepts, SIGMOD Record 21 (3) (1992) 35–43.
- [27] S.-Y. Noh, The formalization of parametric data model in XML schema for temporal data, in: Proceedings of the International Advanced Database Conference, San Diego, CA, USA, 2005, pp. 64–68.
- [28] D.D. Chamberlin, J. Robie, D. Florescu, Quilt: an xml query language for heterogeneous data sources, in: Proceedings of the Third International Workshop on the Web and Databases, 2000, pp. 53–62.
- [29] S.-Y. Noh, S.K. Gadia, A parametric framework for implementing spatiotemporal databases, in: Proceedings of the Second International Conference on Computer Science and its Applications, San Diego, CA, USA, 2004, pp. 312–319.
- [30] S.-Y. Noh, S.K. Gadia, An XML-based framework for temporal database implementation, in: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning, Burlington, VT, USA, 2005, pp. 180–182.
- [31] W3C, Document object model (DOM), Website: <http://www.w3.org/DOM>, April 2005.
- [32] C.-C. Kanne, G. Moerkotte, Efficient storage of XML data, in: Proceedings of the 16th International Conference on Data Engineering, San Diego, CA, USA, 2000, p. 198.
- [33] S. Ma, Implementation of a canonical native storage for XML, Master's thesis, Department of Computer Science, Iowa State University, December 2004.
- [34] D. Barbosa, et al., ToXgene: an extensible template-based data generator for XML, in: Proceedings of the Fifth International Workshop on the Web and Databases, Madison, WI, USA, 2002, pp. 49–54.
- [35] IBM XML Generator, XML generator, Website: <http://www.alphaworks.ibm.com/tech/xmlgenerator/>, December 2004.