Available online at www.sciencedirect.com

ScienceDirect

Journal of Computer and System Sciences 74 (2008) 170–195

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

www.elsevier.com/locate/jcss

XML data update management in XML-enabled database

Eric Pardede [a,*], J. Wenny Rahayu [a], David Taniar [b]

[a] *Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, VIC 3083, Australia*
[b] *Clayton School of Information Technology, Monash University, Clayton, VIC 3800, Australia*

Received 15 June 2006; received in revised form 31 October 2006

Available online 24 April 2007

Abstract

With the increasing demand for a proper and efficient XML data storage, XML-Enabled Database (XEnDB) has emerged as one of the popular solutions. It claims to combine the pros and limit the cons of the traditional Database Management Systems (DBMS) and Native XML Database (NXD). In this paper, we focus on XML data update management in XEnDB. Our aim is to preserve the conceptual semantic constraints and to avoid inconsistencies in XML data during update operations. In this current era when XML data interchange mostly occurs in a commercial setting, it is highly critical that data exchanged be correct at all times, and hence data integrity in XML data is paramount. To achieve our goal, we firstly classify different constraints in XML documents. Secondly, we transform these constraints into XML Schema with embedded SQL annotations. Thirdly, we propose a generic update methodology that utilizes the proposed schema. We then implement the method in one of the current XEnDB products. Since XEnDB has a Relational Model as the underlying data model, our update method uses the SQL/XML as a standard language. Finally, we also analyze the processing performance.
© 2007 Elsevier Inc. All rights reserved.

Keywords: XML data storage; XML-enabled database; XML update; Constraints; Schema; SQL/XML

1. Introduction

With the increasing importance of eXtensible Markup Language (XML) data interchange, the users have to make a crucial decision regarding the repository choice. Many database users prefer to store their XML data in an established database system (DBMS) such as Relational Database (RDB) [17,40], Object Oriented Database (OODB) [16,25] or Object-Relational Database (ORDB) [24,41]. In these DBMSs, the XML data is restructured into their specific storage unit, such as tables in RDB/ORDB and classes in OODB. Another group of users claim that an XML data repository has to facilitate the tree-nature of XML data and the current DBMSs are not sufficient for this purpose, and subsequently propose the use of Native XML Database (NXD) [21,29]. NXD are claimed to be built-from-scratch [4,43] to meet the requirement of XML data. Since they are considerably in their infancy, NXD are not able to compete

* Corresponding author. Fax: +61 3 9479 3060.
E-mail addresses: e.pardede@latrobe.edu.au (E. Pardede), w.rahayu@latrobe.edu.au (J.W. Rahayu), david.taniar@infotech.monash.edu.au (D. Taniar).

0022-0000/$ – see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2007.04.008

with DBMSs that are already in their mature phase of establishment; moreover, there is still a significant number of database users not willing to abandon years of investment and development in already established DBMS.

There has been a recent emergence of XML-Enabled Databases (XEnDB). XEnDB is a database that is built on top of a well-established database to utilize its mature technology with some adjustments to facilitate XML, and is subsequently an obvious choice for XML data storage due to the maturity of the underlying DBMS technology as well. This database contains extensions for transferring data between XML documents and the data structures of their underlying database [3]. Therefore, in this paper, we are focusing on the use of XEnDB as our underlying XML data storage.

Many researchers have worked on the use of established DBMS for XML data storage [32,38]. Recently, we have witnessed an increasing amount of research into the usage of NXD for an XML repository. On the other hand, we cannot find many works on XEnDB storage, perhaps due to a missing general data model for XEnDB. Despite the lack of this data model, ISO/ANSI has been working on a standard for XEnDB. This standard, which is called SQL/XML [24], serves the same purpose as SQL, OQL or XQuery. Similar to these three languages, SQL/XML provides capabilities to store and retrieve the data. However, like XQuery [45], SQL/XML does not provide the full capability for data update. Therefore, in this paper we focus on data update using XEnDB.

Updating XML data is a research topic that is still in its infancy. Even in NXD, update has not been fully implemented [4]. In addition, there is no guarantee that the updated document maintains its original conceptual constraints. Not until recently have we found several works that are dedicated to preserving the constraints of an XML document during update in NXD [33]. However, to the best of our knowledge, no work to date has investigated the preservation constraint during XML update in XEnDB.

This fact is the motivation of our work. Without a constraints preservation mechanism, an updated XML data can become invalid and will not conform to its original XML schema, which is the logical model that defines the properties of the XML data, and consequently incur data inconsistency and redundancy [13]. As XML is now seen as the foundation of e-commerce, it is crucial that some mechanisms be in place to avoid any data inconsistency and redundancy during transactions [31].

The work presented in this paper is part of a larger project on XML update methodologies, which consists of three stages. Stage one focuses on the update methodologies in an Object-Relational Database using different complex types. Stage two focuses on the XML update methodologies in Native XML Database. This paper presents the results of the final stage, the XML update methodologies in XML-Enabled Database. The results of the previous stages have been reported in [34] and [33] for stages one and two respectively.

Our proposal in this paper follows the steps that can be grouped into three parts (see Fig. 1). The first part is the *transformation phase*; the second part is the *update mechanism phase*; and the third part is the *implementation and evaluation phase*.

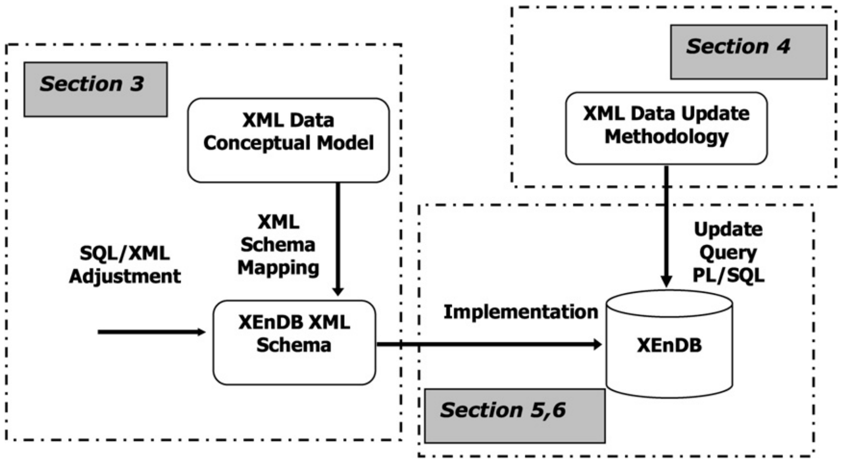


Fig. 1. Our proposed work.

Phase 1: We first identify different conceptual constraints that can exist in XML data. We will map these constraints into a logical model, which takes form as a schema. The result of this transformation will be adjusted with SQL/XML components as the standard for XEnDB.
Phase 2: We propose XML Update methodologies that preserve various data constraints.
Phase 3: We then implement the constraints preservation and the update mechanism in a selected XEnDB product.

The rest of the paper is structured as follows. In Section 2, we briefly provide a motivating example that shows how XML constraint preservation is necessary. In this section we also summarize related works on XML Updates. In Section 3, we provide an overview of our classification on XML constraints and how they are captured in a schema for XEnDB. Next, in Section 4, we propose the update methodologies. We provide an analysis of the computational complexities in Section 5. In Section 6, we present the case study that shows how the proposed methodologies are implemented. Finally, we will conclude our work in Section 7.

2. Related work and motivation

2.1. Related work on XML update

XML update has become increasingly important due to fact that the usage of XML has shifted from being merely a flexible publishing language to a data format. Many works have investigated the importance of efficient XML storage, indexing and query processing [24,26,29]. However, it is widely known that update remains the weakness in many XML databases [4,43].

It is very understandable that the XML communities expect the research initiative on XML update to come from the XML main standard body, the World Wide Web Consortium (W3C). The first draft of the requirement was released in February 2005, and the latest draft was released in June 2005 [46]. Another work that has investigated the XML update was done by XML DB Initiative [28]. This work was intended to propose an XML Update Language (XUpdate) that could be used by any XML Database. Unfortunately, the work has been terminated without the conformance of all XML communities.

Even though we have seen some attempts from the standardization body to unify the XML update process, most existing works have put forward proposals based on the implementation environment.

2.1.1. XML update in database management systems (DBMS)

The advantages of using an established DBMS for XML storage is the full database capability. One of the advantages is full support for update processes. Different manipulation techniques have been implemented by DBMS products and, once we store the XML in a DBMS, we should be able to utilize the update processes.

When people use established DBMS for XML storage, it is very likely that they expect to use the same language for updating both the simple data and the XML data. This is possible since, during the storing process, the XML data will be transformed into the underlying data format such as tables and classes. Some of the works that fall into this category are [17,40] for RDBMS, [19] for OODBMS and [47] for ORDBMS.

XML update using the basic DBMS language is straightforward and, therefore, it is popular. However, there are two issues in this option. First, the DBMS language is standardized to cater for a specific data format. For example, SQL is developed to deal with tuples and tables. Even though we can use SQL, it does not exactly match the nature of XML query language. Second, we have to deal with the transformation process during the storage. If the mapping of XML into tuple or object is not correct, the update will not be optimal. Depending on the complexity of the XML data, the transformation can be expensive and some of the XML semantic might be diminished.

Some works propose methods to translate XML query languages (and update for that matter) into basic query languages such as SQL. These works seek to address the problem mentioned before, which is the different nature of the XML query language and the basic DBMS query languages.

This option covers the expressive power of the XML query languages. Many proposals [1,10,11,23,45] have been issued regarding the adjusted query languages. However, these are mostly discussed only within the research community. In addition, none of them covers whole update operations or considers the constraint preservation. In addition, the adjusted query languages are not standard languages, and thus, will be unlikely to gain support from the majority of the XML community.

2.1.2. XML update in native XML database (NXD)

In established DBMS, the update process is usually straightforward. Therefore, we cannot find many works that consider discussing the issue. However, for NXD the XML update, it is still an open problem and provides no standard that can be used as the guideline. There are several strategies for updating XML documents in NXD [4,43].

Many NXD products use their own proprietary update languages that will allow XML updating within the server. They include Lore [18] and SODA [48]. Despite their update ability, the language is not a standard language that can be accepted and used by the whole XML community. It is observed that more recent products no longer use proprietary language. The conformance to standard XML query languages like XQuery has become a basic requirement.

Other products use a special language called XUpdate [49]. They include eXist [2,30], and dbXML [9]. The XUpdate specification defines the syntax and semantics of the language [28]. It is designed to be used independently of any kind of implementation. In the implementation, XUpdate is usually used with another language for retrieval purposes.

The other strategy that is followed by most NXD products is to retrieve the XML document and then update using XML Application Programming Interface (API). After update, the document is returned to the database. They include TIMBER [21] and Berkeley dbXML [42]. This option is the most widely used option for XML update. However, the main reason behind this is the current limitation of XML query languages for update purposes. In addition, having a separate API for update is costly.

The last option for XML update in NXD is by embedding the update processes into XML language. This is the latest development and the interest is growing. The first work that started the idea was [44], which embedded simple update operations into XQuery and thus could be used for any NXD that supported this language. Since then, few works use [44] as the template for their proposals [22,27]. Nonetheless, even with this proposal, there is a basic issue that remains unresolved. We do not know how the update operations can affect the semantic correctness of the updated XML documents.

2.1.3. XML update in XML enabled database (XEnDB)

In this paper, we focus on the XML update in another database family, XML-Enabled Database. This database is developed using an established-DBMS with many XML-related features extension.

Unfortunately, there are not many works on XEnDB as XML storage. The reason behind this is that XEnDB products have different data models and thus, the modeling and implementation will be different as well [5]. We cannot find any work that discusses the update methodologies in this database family. Of course, there is some basic update operation syntax in some of the product manuals. However, they cover only very primitive update operations without considering the constraints of the data.

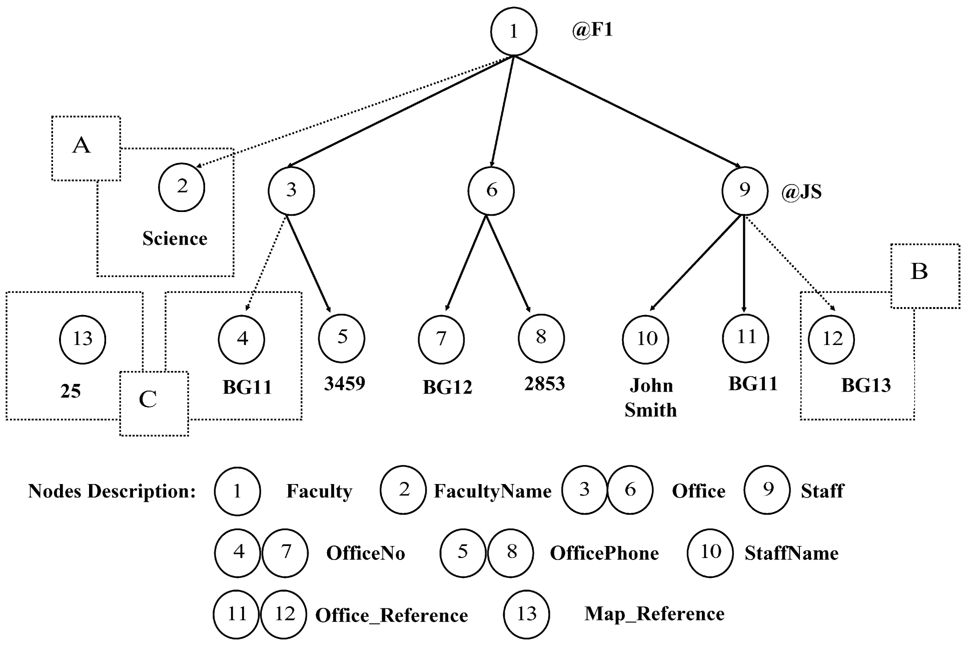
[37,39] discuss a different technique for retrieving XML TYPE data. Oracle 10g provides a different built-in function to comply with the latest SQL/XML Standard [12,15,20]. However, it covers only a very limited update operation and the update still has to utilize other built-in functions. For example, to update the text content of an XML element, the user has to use *text*( ) function in XPath expressions. The discussion does not cover how to carry out safe manipulation by checking the XML constraints.

DB2 [8] claims that the product can store the XML data as column and collection. For the update operation, they have the DB2 Text Extender that, it is claimed, has the ability to carry out a full-text index update along with automatic synchronization of the text index and the tables [7]. Unfortunately, there is no clear information on how the product performs the synchronization or whether it really maintains the full constraints of the XML.

From the survey, we have found that there is no unified update technique for XEnDB. A standard technique for update in XEnDB is needed, each for a different data model foundation such as Relational Model, Object-Oriented Model, Nested Relational Model, etc.

2.2. Motivating example

We have discussed existing works on XML update and how they have not considered one important aspect in data manipulation, which is the consistency of constraints. Now we can provide a motivating example that can show why this is an important issue. We use the sample XML document in the following Fig. 2.



XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"                A
elementFormDefault="qualified">
  <xs:element name="Faculty">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="FacultyName" type="xs:string" minOccurs="1"/>
      <xs:element ref="Subject" maxOccurs="unbounded"/>
      <xs:element ref="Staff" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="FacultyID" type="xs:string" use="required"/>
  </xs:complexType>
  </xs:element>
  <xs:complexType name="OfficeType">                                        C
    <xs:sequence>
      <xs:element name="OfficeNo" type="xs:string" minOccurs="1"/>
      <xs:element name="OfficePhone" type="xs:string"maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Office" type="OfficeType"/>
  <xs:complexType name="StaffType">
    <xs:sequence>
      <xs:element name="StaffName" type="xs:string"/>
      <xs:element name="Office_Reference" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="StaffID" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:element name="Staff" type="StaffType"/>
  <key name=”OfficeNo_Key”>
    <xs:selector xpath=”Faculty/Office”/>
    <xs:field xpath=”OfficeNo”/>
  </key>
  <keyref name=”Office_Reference” refer=”OfficeNo_Key”>
    <xs:selector xpath=”Faculty/Staff”/>                              B
    <xs:field xpath=”Office_Reference”/>
  </keyref>
</xs:schema>
```

Fig. 2. XML document simple update example.

There are three types of update operations. *Deletion* is the process of removing a target from the database. *Insertion* is the process of adding a target into the database. *Replacement* is the process of exchanging a target in the database with another one.

As an example of deletion, in Fig. 2 we want to delete a Faculty Name node "Science". This is denoted with rectangle A. The primitive operation will allow this process. However, if there is a constraint that a faculty must have at least one name, as is shown on the schema, this deletion creates an invalid XML data. Without constraint checking, we will end up with an XML document with violated cardinality constraint.

A classic example of the insertion problem is shown in Fig. 2 with rectangle B. It is noted on the schema that the Office Reference in Staff refers to Office Number in Office node. We want to insert a reference node with content "BG13," which is not a valid office number. In primitive insertion, this process can be done and we end up with a node that refers to an invalid node. For safe insertion, we need to check all target instances to ensure that the referred node exists.

As an example of replacement, in Fig. 2 we want to replace the node Office Number into node Map Reference. This is denoted with rectangle C. In primitive update, this process will be valid. However, it actually violates the original schema. Now, an office does not have office number. On the other hand, it has another node type. In addition, it leaves other nodes that refer to office number "BG11" with invalid destination. For safe replacement, we will have to check the schema and all other references beforehand.

We have described three update operations that can be performed to XML documents and their consequences to constraint violations. It is clear that we require new methodologies to accommodate safe update operations. Safe update operations mean that the methodologies not only allow the update operations, but also allow the users to check the conceptual constraints of the document. In the next section, we will show how the constraints are classified for this work and how to represent them in the schema.

3. XML constraints: An overview

XML constraints are commonly categorized based on the relationship inside the XML document. They are association, aggregation and inheritance relationship. Further discussions of the classification with an example can be found in [34].

The constraints related to an association relationship are: number of participant type, referential integrity, cardinality, and adhesion. The constraints related to an aggregation relationship are: cardinality, adhesion, ordering, homogeneity and share-ability. Finally, the constraints related to an inheritance relationship are: exclusive disjoint constraint and the number of ancestor.

In the first impression, the XML constraint classification does not conform to the traditional XML constraints classification. Therefore, to avoid misinterpretation of XML constraints, we need to clarify our XML constraints classification in the first section.

3.1. XML semantic constraints

Similar to any other data structure, XML data contains many traditional constraints such as key constraint, inclusion constraint, inverse constraint, type constraint, etc. [6]. These constraints are specifically applied to the tree nature of XML data. In addition to these classical constraints, other constraints such as the path constraint are specifically unique to XML data. [14] discusses the XML constraints from the tree-structure perspective.

In this paper, we do not focus on this type of constraint. Rather, we focus on the XML semantic constraints. These constraints are user-defined constraints, which can be defined in the XML schema or in the XML instance. These constraints define how a subset of XML data relates with other subset(s) of XML data.

Even though the constraints that are presented in this paper do not use the traditional XML constraint, the proposed classification does not contradict the traditional XML constraints. The semantic constraints still have to conform to the XML tree structure. For example, the association and aggregation relationship constraints (see Fig. 3) apply the XML key constraint, referential constraint as well as path constraint.

This simple XML tree has a root node of type *Faculty*. This root node has many children nodes at level 1 of the XML tree. It has a *Faculty Name*, many *Office* nodes and a *Staff* node. The last two nodes are complex elements and thus, have more children at level 2 of the XML tree.

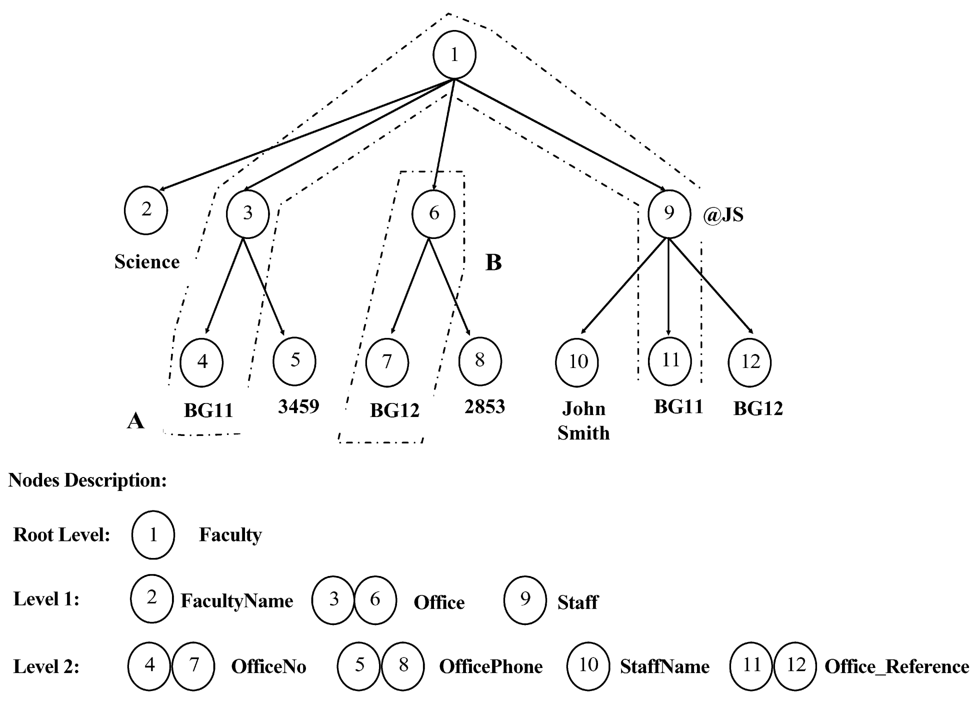

1
2
Science
3
4
BG11
5
3459
6
B
7
BG12
8
2853
9
@JS
10
John Smith
11
BG11
12
BG12
A
Nodes Description:
Root Level: 1 Faculty
Level 1: 2 FacultyName 3 6 Office 9 Staff
Level 2: 4 7 OfficeNo 5 8 OfficePhone 10 StaffName 11 12 Office_Reference

Fig. 3. XML semantic constraints example.

This XML tree contains traditional XML constraints. Key constraint, which determines the element/attribute that uniquely differentiates an element, is defined in *FacultyName*, *OfficeNo* and *StaffID* attribute. Path constraint determines the path of one node traversed from the root node. For example, the path to the key node of the first office is FACULTY/OFFICE[1]/OFFICENO. Inverse constraint determines that every staff occupies office(s) and every office is occupied by staff.

The way we capture these constraints is through the semantic relationship between nodes. XML tree subset *A* shows an association relationship. A *Staff* has element *Office_Reference* (FACULTY/STAFF/OFFICE_REFERENCE), which refers to an *OfficeNo* (FACULTY/OFFICE/OFFICENO). In another word, an *OfficeNo* is referred by an *Office_Reference*. This referential constraint in an association relationship captures many traditional tree constraints such as path constraints, key constraint, reference constraint and inverse constraint.

Another example, XML tree subset *B* shows an aggregation relationship. An *Office* is a child node of a root node *Faculty* and it has one and only one *OfficeNo* child. In other words, there is an inclusion constraint of the *OfficeNo* in the *Office* node. This aggregation relationship captures many traditional tree constraints such as path constraints and the inclusion constraint.

These two examples demonstrate how we capture different traditional XML tree constraints through a higher level of classification, i.e. XML semantic constraints.

There are two reasons for classifying XML constraints based on the semantic. Firstly, the semantic constraints, which are based on the object-oriented concept, have relationship types that are easy to understand and can easily be used to describe user requirements. For example (see Fig. 3), rather than using paths to describe relations among nodes *Office*, *OfficeNo* and *OfficePhone* (FACULTY/OFFICE/OFFICENO and FACULTY/OFFICE/OFFICEPHONE), we can easily describe an *Office* as the aggregation of *OfficeNo* and *OfficePhone*.

Secondly, semantic classification is richer in terms of identifying many constraints that are not explicitly defined in traditional XML constraints. For example, the exclusion disjoint constraint is a constraint that determines whether a complex node can inherit properties to one or more complex nodes. This constraint is unique to the inheritance relationship and without classification based on a semantic relationship; we can easily overlook this constraint.

3.2. Preserving constraints in XML schema

Now we have identified the constraints, we can propose a way of capturing them in the logical schema. Preservation of XML Conceptual constraints cannot be separated from the storage of the data. In the last three years, we have been working on the project of preserving the constraints in different data storage. As the result, in [34] we map the XML constraints into ORDB schema and implement the data in various complex ORDB types. In [33] we identify the constraints in XML Schema and implement it in NXD.

For XEnDB, the schema will be XML Schema with SQL/XML annotation. For illustration purposes, we use the XML document shown in Fig. 4 for association and aggregation relationship constraints. We use XML document shown in Fig. 5 for inheritance relationship constraints. We will not discuss mapping for each constraint because of the lengthy discussion. In addition, this is only a part of this paper's contribution. More of the transformation can be found in [35].

The preservation of the association relationship constraint is very straightforward. In XML Schema, we can determine the "minoccurs" and "maxoccurs" after an element declaration. It is general practice that these constraints are not used for attribute. A particular attribute will appear, at most, once in an object/instance.

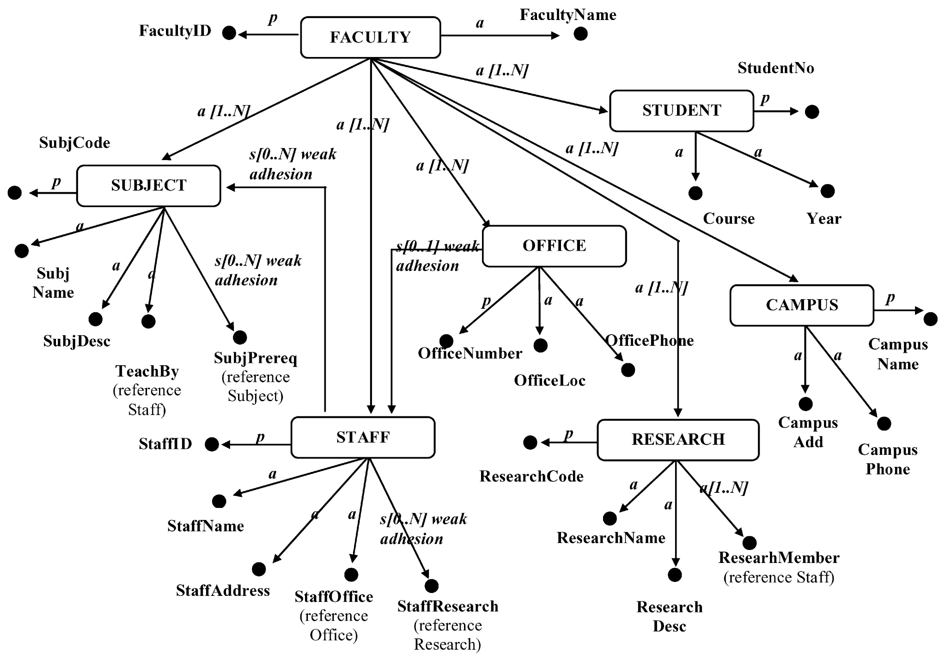


Fig. 4. XML document tree Example 1.

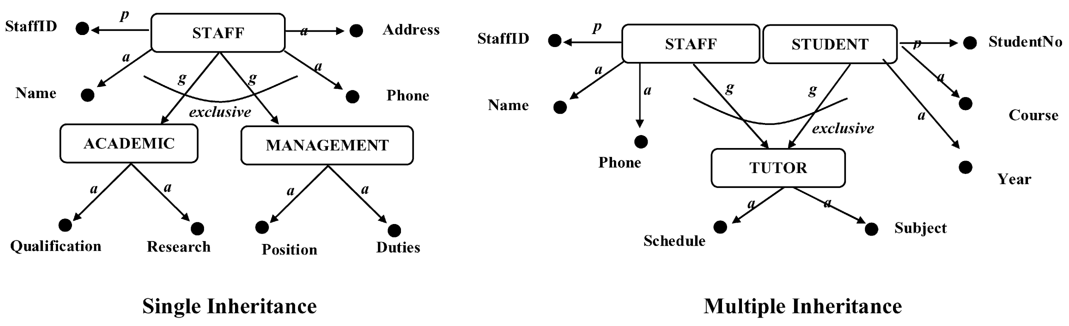


Fig. 5. XML document tree Example 2.

For example, a faculty has to have one, and only one, name. If we use an element, we can determine the cardinality constraints as is shown below. Faculty also has to have one, and only one, ID. For an attribute, we do not use the same cardinality constraints. Instead, we just employ "use" constraint to "required."

```
<xs:element name = ''Faculty''>
  <xs:complexType>
    <xs:element name=''FacName'' type=''xs:string''
        minoccurs=''1'' maxoccurs=''1''>
..................
  <xs:attribute name=''FacID'' type=''xs:string'' use=''required''/>
</xs:complexType>
</xs:element>
```

Now, we have to transform the XML Schema into XEnDB Schema. First, we add the namespace that contains the annotations. In this case, the namespace *xdb* is available at http://www.oracle.com/xdb. All elements with "maxoccurs" larger than one will be implemented as collection in XEnDB. In the schema below, we define *xdb:storeVarrayAsTable* equals to "true." This global XML annotation tells the database to store all varray as nested tables. It can speed up the queries on XML elements [39].

```
<xs:schema xmlns:xs=''http://www.w3.org/2001/XMLSchema''
  xmlns:xdb=''http://www.oracle.com/xdb''
  xdb:storeVarrayAsTable=''true''>
```

When XEnDB maps the XML schema to tables, they will give default names to the tables and attributes. These names are not easy to remember and they do not represent the content of the data. Therefore, we can specify the name of the tables, types and attribute name. The schema below shows the element Faculty with the schema annotations.

```
<xs:element name=''Faculty'' xdb:defaultTable=''FacultyTable''>
  <xs:complexType name=''FacultyType'' xdb:SQLType=''XDB\_FacultyType''>
    <xs:sequence>
      <xs:element name=''FacID'' type=''xs:string'' xdb:SQLName=''FacID''
           xdb:SQLType=''VARCHAR2''/>
      <xs:element name=''FacName'' type=''xs:string'' xdb:SQLName=''FacName''
           xdb:SQLType=''VARCHAR2''/>...
    </xs:sequence>
  </xs:complexType>
```

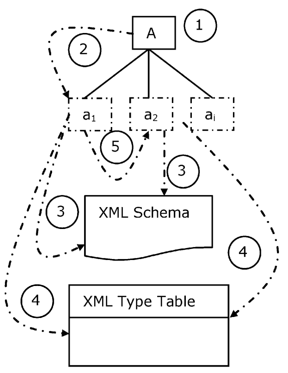
Note that in a table, there is no difference between attribute and elements. Thus, we need to convert all XML attributes into elements. It is necessary so we can embed constraints to it later on.

4. Proposed methodologies

In this section, we propose the update methodology for each relationship to capture the conceptual constraints. The update can be classified into deletion, insertion and replacement. The target of the operations can be either an attribute or an element. We provide algorithms that describe how we maintain different constraints. These algorithms are generic steps that are applicable for any XEnDB products. Obviously, various XEnDB products will most likely implement these steps differently.

4.1. Preserving association relationship

For an association relationship, we are concerned with the updates that involve attribute and elements that have a role as KEY or KEYREF. Other constraints checking (adhesion and cardinality) will be only in the context of KEY/KEYREF. The checking for adhesion and cardinality in non-KEY/KEYREF will be discussed in Section 4.2, since



```
Operation 1 (Read the Element)
FOR ALL attribute and leaf element
  Operation 2 (Read the att.and leaf ele.)
  Operation 3 (Check the XML Schema)
  IF target is Key
    THEN
      Operation 4 (Check the XML Type Table)
      IF it is a duplicate
        THEN Stop Insertion
        ELSE ( )
    ELSE Operation 5 (Move to the next
               att./leaf ele.)
```

Algorithm 1. Insertion update for KEY target.

these constraints also exist in an aggregation relationship. For each of the operations, we can select three maintenance operations, which are deletion, nullify and cascade.

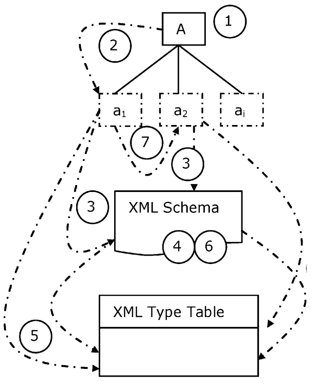
In insertion, we need to check for KEY and KEYREF target. For key insertion, we have to ensure that there is no duplication of unique values. For KEYREF insertion, we need to check whether the new KEYREF actually refers to an existing key.

Algorithms 1 and 2 show the checking mechanism for target KEY and KEYREF respectively. Note that some of the maintenance operations might not be practical. An example is the cascade strategy for KEYREF insertion. If the referred key does not exist, we have to get the information before we can perform the KEYREF insertion. Such information is not always available.

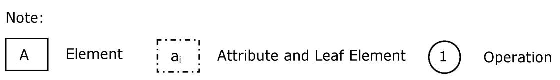
Next, deletion is the simplest update operation, especially if we want to delete an attribute or an element that contains simple data. In this case, we do not need to perform the checking on the node values. However, if the target also has roles as KEY/KEYREF, the checking on values is required. This is in order to maintain the referential integrity constraint.

Algorithm 3 shows the deletion checking mechanism for key target. KEYREF node on the other hand can be treated like simple data content. Before the deletion of a key can be performed, we need to check whether the target key instance has KEYREF referred to it since we do not want to leave a dangling reference in the document. By having the KEY/KEYREF information in the schema, the implementer can easily check which KEYREFS refer to a specific KEY.

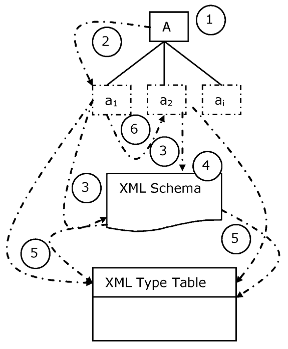
Algorithm 3 shows the checking mechanism if the target is a KEY regardless of whether it is an attribute or an element. We employ three possible maintenance strategies if the target is indeed a KEY and being refereed by other



```
Operation 1 (Read the Element)
FOR ALL attribute and leaf element
  Operation 2 (Read the att. and leaf element)
  Operation 3 (Check the XML Schema)
  IF target is KeyRef
    THEN
      Operation 4 (Check the Key node being
                   referred)
      Operation 5 (Check the Adhesion and
                   Cardinality constraints)
        IF the NEW destination NOT exists
          THEN Stop Replacement
          ELSE ()
      Operation 6 (Check the Key Value in XML
                   Type Table)
        IF the adhesion/cardinality constraint
           is violated
          THEN Stop Replacement
          ELSE ()
    ELSE Op.7 (Move to the next att./leaf
                   ele.)
```



Algorithm 2. Insertion update for KEYREF target.



```
Operation 1 (Read the Element)
FOR ALL attribute and leaf element
 Operation 2 (Read the att. and leaf ele.)
 Operation 3 (Check the XML Schema)
 IF target is Key
  THEN
   Operation 4 (Check the Keyref of the Target)
   Operation 5 (Check the XML Storage keyref
               instance)
   IF the target is being referred
    THEN
     CASE "restrict": Stop deletion
     CASE "nullify": Replace the keyref instance
                     to NULL
     CASE "cascade": Delete the referring
                     instance
    ELSE ( )
  ELSE Op.6 (Move to the next attribute/leaf
            ele.)
```

Note:

A Element ; aᵢ Attribute and Leaf Element ; 1 Operation

Algorithm 3. Deletion update for KEY target.

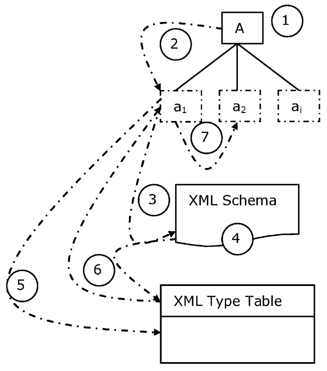
nodes. The deletion will be canceled if the strategy is restricted. Otherwise, the deletion will be performed after some preliminary actions such as nullifying the KEYREF value for nullify strategy and deleting the keyref for the cascade strategy.

The last update operation is the replacement. This operation can be seen as a combination of deletion and insertion. Therefore, the previous algorithms can be used for the checking mechanism. The only concern is whether to use the OLD or the NEW value during the checking. Algorithms 4 and 5 show the checking mechanism for KEY and KEYREF target respectively.

4.2. Preserving aggregation relationship

In this section, we show the proposed methodologies to capture the document constraints associated with an aggregation relationship. Notice that we can also encounter the KEY/KEYREF as a "part" component of a "whole" component node. However, since the preservation of this constraint has been discussed in the previous section, we will not repeat KEY/KEYREF "part" component in this section. Every time we discuss an attribute or an element, we are not concerned with their functionality in the referential mechanism.

Finally, for an aggregation relationship constraint, we do not have a different strategy maintenance as we do for the association relationship constraints. All operations that have violated the schema will be restricted. No nullify or cascade options are available.

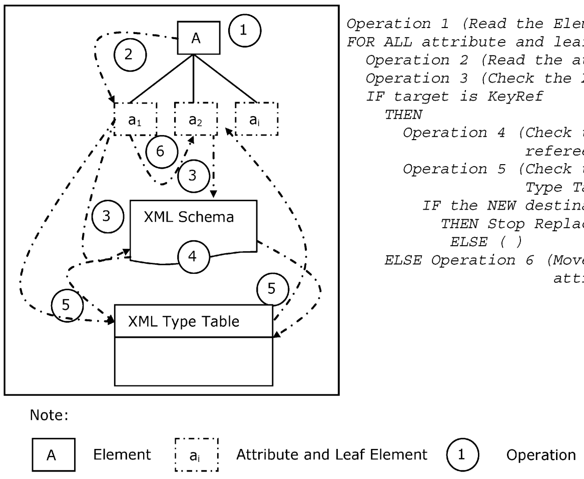


```
Operation 1 (Read the Element)
FOR ALL attribute and leaf element
 Operation 2 (Read the att. and leaf ele.)
 Operation 3 (Check the XML Schema)
 IF target is Key
  THEN
   Operation 4 (Check the Keyref of the Target)
   Operation 5 (Check the XML Storage)
    IF the NEW target is a duplicate
     THEN Stop Insertion
     ELSE
      Operation 6 (Check the XML Storage keyref
                  instance)
       IF the target is being referred
        THEN
         CASE "restrict": Stop deletion
         CASE "nullify": Replace the keyref
                         instance to NULL
         CASE "cascade": Replace the keyref
                       instance to the NEW target
        ELSE()
  ELSE Op.7 (Move to the next att/leaf ele.)
```

Note:

A Element ; aᵢ Attribute and Leaf Element ; 1 Operation

Algorithm 4. Replacement update for KEY target.



```
Operation 1 (Read the Element)
FOR ALL attribute and leaf element
  Operation 2 (Read the att. and leaf ele.)
  Operation 3 (Check the XML Schema)
  IF target is KeyRef
    THEN
      Operation 4 (Check the Key node being
                   refereed)
      Operation 5 (Check the Key Value in the XML
                   Type Table)
        IF the NEW destination NOT exists
          THEN Stop Replacement
          ELSE ( )
    ELSE Operation 6 (Move to the next
                      attribute/leaf element)
```

Algorithm 5. Replacement update for KEYREF target.

Algorithm 6 shows the proposed checking method for the insertion operation. Algorithms 6-1 to 6-4 show in detail the procedures for each constraint.

Algorithm 7 shows the proposed checking method for the deletion operation. Algorithms 7-1 and 7-2 show the detail procedures for each constraint. We differentiate these into element and attribute, since the constraints are declared differently.

Note that there is no checking required for homogeneity or ordering constraint. This is because these constraints will not be affected by the deletion operation. As for the share-ability constraint, the deletion mechanism will follow the checking mechanism in an association relationship. This is because in XML Schema we represent the shareable element as a separate type and use the KEY/KEYREF mechanism.

Finally, unlike an association relationship, we do not need an additional algorithm for replacement update. This is because violation of the schema occurs during replacement of the node with simple content. For example, the number of attributes or elements will be the same and therefore the cardinality and adhesion constraints are not affected. The position of the element will remain the same and therefore the ordering and homogeneity constraints are still met.

4.3. Preserving inheritance relationship

In this section, we show the proposed method for capturing the document constraints associated with an inheritance relationship. In our transformation methodology, we preserve the exclusive disjoint constraint through a new class type. We also associate the super-class and sub-class through the key of this new class.

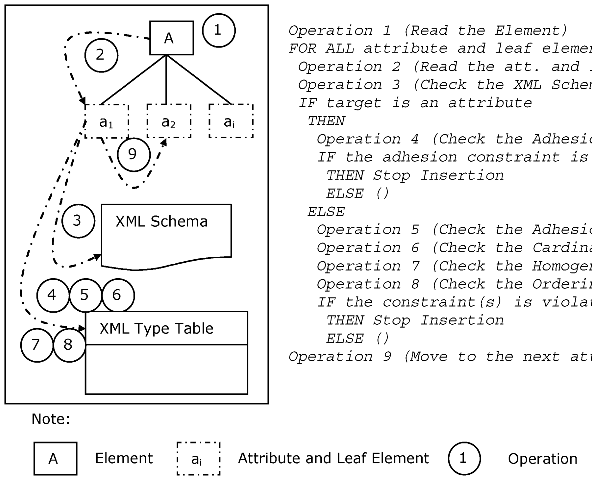
Note that an inheritance relationship can only exist for the element. Therefore, no checking mechanism if the update target is an attribute. Finally, similar to the association relationship constraint, we can apply three different maintenance strategies, which are restrict, nullify and cascade for the inheritance relationship. This is because the relationship between the super-type and sub-type keys resembles the KEY/KEYREF in the association relationship.

For insertion, we have to check the sub-class instance in order to avoid the choice constraint violation. On the other hand, we do not have to check the insertion of a new super-class instance. Algorithm 8 shows the mechanism for insertion update, which is applicable both for single and multiple inheritances.

In Algorithm 8, we have action only for restrict and nullify strategies. If there is choice constraint and the target duplicates the super-type key, we have to stop the insertion for the restrict strategy or replace the new super-type key value with NULL for the nullify strategy.

Algorithm 9 shows the mechanism for the deletion in an inheritance relationship. It can be applied both for single and multiple inheritances. The difference lies in the target of the checking.

The checking is required if the deletion is conducted upon a super-class instance, where we have to check whether there is any sub-class instance with the same key node. We are not concerned with the deletion of a sub-class instance because the same key node might exist in another sub-class instance or in the super-class instance.



```
Operation 1 (Read the Element)
FOR ALL attribute and leaf element
 Operation 2 (Read the att. and leaf element)
 Operation 3 (Check the XML Schema)
 IF target is an attribute
  THEN
   Operation 4 (Check the Adhesion const.)     [1]
   IF the adhesion constraint is violated
    THEN Stop Insertion
    ELSE ()
  ELSE
   Operation 5 (Check the Adhesion const.)     [1]
   Operation 6 (Check the Cardinality const.)  [2]
   Operation 7 (Check the Homogeneity const.)  [3]
   Operation 8 (Check the Ordering const.)     [4]
   IF the constraint(s) is violated
    THEN Stop Insertion
    ELSE ()
Operation 9 (Move to the next att./leaf element)
```

Algorithm 6-1. Adhesion Constraint Checking

```
IF the Target is an attribute
  THEN
    Check the element where the attribute is attached
    IF the attribute already exists
      THEN
        Stop the insertion
      ELSE insert the target
```

Algorithm 6-2. Cardinality Constraint Checking

```
FOR ALL elements in the target
  Check the maximum occurrence constraint
  IF the constraint exists
    THEN
      Check the Instance occurrence
      IF the Instance occurrence > (maximum occurrence + 1)
        THEN
          Stop the insertion
        ELSE()
    ELSE()
```

Algorithm 6-3. Homogeneity Constraint Checking

```
Check the homogeneity constraint
IF the element is under homogeneous constraint
  THEN
    Check the parent element of the target element
    IF the parent element has another child of different type
      THEN
        Stop the insertion
      ELSE ()
  ELSE ()
```

Algorithm 6-4. Ordering Constraint Checking (cont)

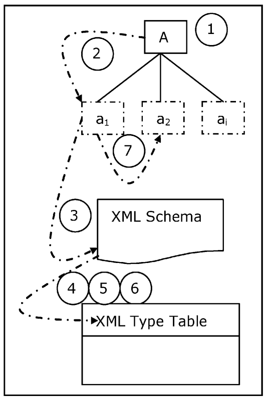
```
Check the ordering constraint
IF the element has sequence constraint
  THEN
    Locate the previous element
    -- when insertion is performed we use this information
ELSE ()
```

Algorithm 6. Insertion update for constraints in aggregation relationship.

Algorithm 9 shows the two maintenance strategies. For the restrict strategy, we cancel the deletion if the super-type key exists in another element. For the cascade strategy, we have to delete all elements with the same key value before we delete the target.

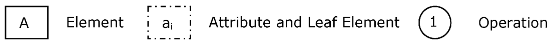
For the replacement operation, we have to check whether the replacement of a super-key instance is consistent in different sub-type instances. Algorithm 10 shows the mechanism for replacement update, which is applicable both for single and multiple inheritances. This algorithm is applicable for sub-class replacement.

Algorithm 11 shows the mechanism for the replacement update, which is applicable both for single and multiple inheritances if the target is a super-class. It shows two maintenance strategies. For the restrict strategy, we cancel the



```
Operation 1 (Read the Element)
FOR ALL attribute and leaf element
 Operation 2 (Read the att. and leaf ele.)
 Operation 3 (Check the XML Schema)
 IF target is an attribute
  THEN
   Operation 4 (Check the Adhesion const.)    [1]
   IF the adhesion constraint is violated
    THEN Stop Deletion
    ELSE ()
  ELSE
   Operation 5 (Check the Adhesion const.)     [2]
   Operation 6 (Check the Cardinality const.) [3]
   IF constraint(s) is violated
    THEN Stop Deletion
    ELSE ()
Operation 7 (Move to the next att./leaf ele.)
```

Note:



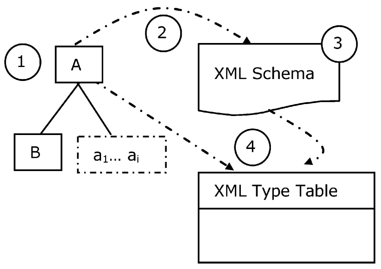
Algorithm 7-1. Adhesion Constraint Checking in Attribute

```
Read the Target
  IF the Target is an attribute
    THEN
      Check the "use" constraint in the schema
      IF the "use" is "required"
      THEN
        Stop the deletion
        ELSE delete the target
```

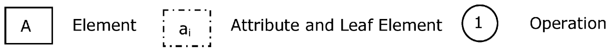
Algorithm 7-2. Adhesion and Cardinality Constraints Checking in Element

```
  FOR ALL elements in the target
    Check the minimum occurrence constraint
    IF the constraint exists
      THEN
        Check the Instance occurrence
        IF the Instance occurrence < (minimum occurrence + 1)
        THEN
          Stop the deletion
        ELSE delete the Target
      ELSE Delete the Target
```

Algorithm 7. Deletion update for constraints in aggregation relationships.

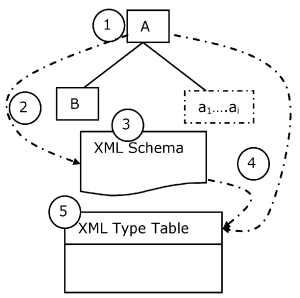


Note:



```
Operation 1 (Read the Element)
Operation 2 (Check the XML Schema)
Operation 3 (Check the inheritance and "choice" constraint)
 IF there is a super-class & "choice" exists
  THEN
   Operation 4 (Check the XML Storage)
    IF the super-class has another sub-class
     THEN Stop Insertion
     ELSE ()
  ELSE ()
```

Algorithm 8. Insertion update for sub-class in inheritance relationship.

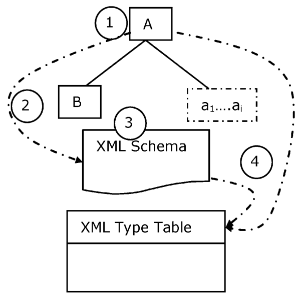


```
Operation 1 (Read the Element)
Operation 2 (Check the XML Schema)
Operation 3 (Check the inheritance)
 IF there is a sub-class(es)
  THEN
   Operation 4 (Check the XML Storage)
    IF the sub-class instance exists
     THEN
      CASE "restrict": Stop the deletion
      CASE "cascade": Operation 5(Delete
                      the sub-classes)
     ELSE ()
  ELSE ()
```

Note:

A Element ai Attribute and Leaf Element 1 Operation

Algorithm 9. Deletion update for super-class in inheritance relationship.

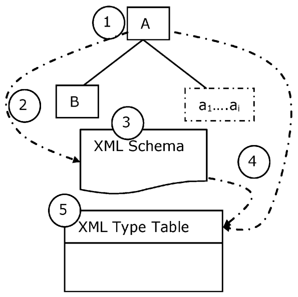


```
Operation 1 (Read the Element)
Operation 2 (Check the XML Schema)
Operation 3 (Check the inheritance and
            "choice" constraint)
 IF there is a super-class & "choice" exists
  THEN
   Operation 4 (Check the XML Storage)
    IF the new target type the same with the
       old type
     THEN ()
     ELSE
      IF the super-class has another sub-
         class
       THEN Stop Replacement
       ELSE ()
  ELSE ()
```

Note:

A Element ai Attribute and Leaf Element 1 Operation

Algorithm 10. Replacement update for sub-class in inheritance relationship.



```
Operation 1 (Read the Element)
Operation 2 (Check the XML Schema)
Operation 3 (Check the inheritance)
 IF there is a sub-class(es)
  THEN
   Operation 4 (Check the XML Storage)
    IF the sub-class instance exists
     THEN
      CASE "restrict": Stop the replacement
      CASE "cascade": Operation 5 (Replace
                      the sub-classes)
     ELSE ()
  ELSE ()
```

Note:

A Element ai Attribute and Leaf Element 1 Operation

Algorithm 11. Replacement update for super-class in inheritance relationship.

deletion if the old super-type key exists in another element. For the cascade strategy, we have to replace the same key value before we replace the target.

5. Evaluation

This section evaluates the impact of the proposed methodologies upon XML update. In this section, we propose a formalization technique that can model the computational complexity in XML updates. This formal model is generic and implementation-independent. The model compares the primitive update and the safe update processes. The safe update in this case is the update that performs an update checking, whereas the primitive update does not. It is important to emphasize that the aim of the evaluation is for comparative study only and therefore, some simplification of the processes is made.

The computational complexity is modeled using different parameter sizes, such as number of documents, size of the target, the position of the target in the instance and the schema complexity. The size and position of the target will be determined by using a simple metric in the XML tree. By alternating these parameters, we can see how the checking mechanisms affect the XML update processes. The model is formed by three units/components: (i) instance searching component (ii) CPU update processing component, and (iii) schema checking component.

$$\boxed{\begin{aligned} \mathit{Total\_Unit} &= \mathit{Inst\_Search\_Unit} + \mathit{Update\_Unit} + \mathit{Schema\_Search\_Unit} \\ &= \sum_{i=1}^{m}\left(\sum_{j=0}^{n}\mathit{Horizontal\_Traversal\_Path} + \sum_{k=0}^{p}\mathit{Vertical\_Traversal\_Path}\right) x S \\ &\quad + \sum_{l=1}^{q}(N_U) x U + \sum_{v=0}^{y}\mathit{Schema\_Traversal\_PathxS} \end{aligned}}$$

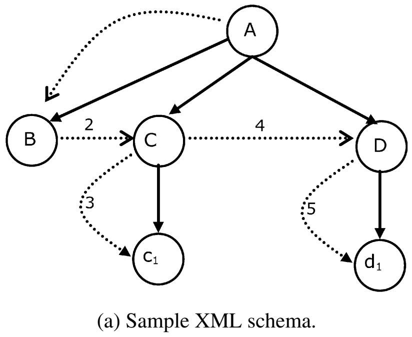
where i is the number of instance document to be searched, j is the number of horizontal traversal path in the searched instances, k is the number of vertical traversal path in the searched instances, l is the number of instances to be updated, N_U is the number of element/node that will be updated in each instance and v is the number of traversal paths in the schema during the constraint checking. Finally S and U represent the processing units for searching and update respectively.

The first component models the resources needed to search for the location of the updated target. This component also depends on the number of search paths. If the update spans over different XML document instances, the number of paths in each document can be different. However, for simplification, in this section we assume the paths are all in the same numbers. This can be justified since our aim is only to compare the primitive update with the safe updates.

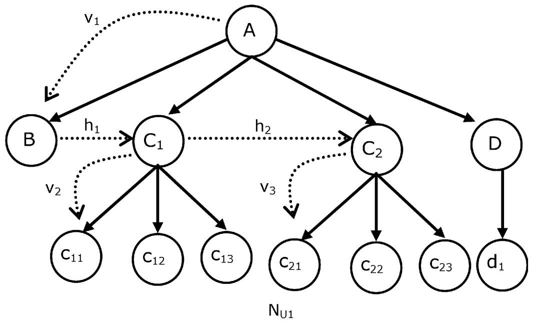
The second element models the resources needed to update the target. It is dependent on the number of document instances and the number of target nodes in each instance. In this section, we assume all instances that have update targets actually have the same number of update target nodes. We also do not differentiate the update unit (U) for different operations. This is because we will compare the same update operation for primitive and safe updates.

The third component models the resources needed to search the schema and find out all constraints associated with the updated instance. This unit is also dependent on the number of search paths. The total number of the paths will be dependent on the size of the schema.

For illustration purposes, we use the following XML document (see Fig. 6). Figure 6(a) shows the sample XML schema and Fig. 6(b) shows the instance for this schema. This XML instance has root node ‘A.’ In the first level of the XML tree, there are four child nodes and they are formed by three different element types. Children numbers 2



(a) Sample XML schema.



(b) Sample XML document.

Fig. 6. Sample XML for performance evaluation.

and 3 are of ‘C’ type. Each of the children has other children and they are shown in the second level of the tree document.

If we want to update node c_{21} for example, Fig. 6(b) shows the search process for finding the target node. Following the formal process model, we assume there is only one document instance with this type in the database and therefore m equals to 1. The total horizontal traversal path (n) and vertical traversal path (p) are 2 and 3 respectively. Since the number of target node is 1, the value of q also equals to 1.

If we have a constraint checking mechanism, we have to perform some traversal in the schema as well and this process is shown in Fig. 6(a). To check all constraints in that schema, we have to traverse to 5 paths and this is the value of y.

In the following sections, we will use the formal process model to compare the processes undertaken, and hence, the computational complexity, for different update operations. In each operation type, we compare the processes of updating various targets both by using our checking update methodologies and by using a primitive update.

5.1. *Insertion evaluation*

In association relationship structure, we have two cases: referencing and dereferencing insertion. For both cases we will use the association relationship between element C and element D or their children nodes (see Fig. 7). We assume that the update occurs in a single document instance. Therefore, the value of m in the formal model is 1.

In the referencing insertion, the user wants to insert new node(s) that happens to refer to another node in the document. As example (see Fig. 7), we want to insert element D_2 and its child d_{21}. D_2 value actually refers to a certain value stored in the first child node of element C.

For primitive insertion, we do not need to check whether the insertion target has a valid reference. The process of insertion incurs traversal process only to the end of the document and the actual insertion operation. For safe update however, we need to check the values of all first child of element C. In this case, since the document has two C elements, we need to check the value of c_{11} and c_{21}. This process incurs additional traversal paths. Only if the target reference exists, are we able to perform the insertion. Also, for safe insertion, there is an additional process for checking the schema (see Fig. 6(a)). If the schema is more complex, there will be more traversal paths to be counted during the constraint checking.

Following the formal model, in primitive insertion there are 3 horizontal paths and 1 horizontal path for the first formal model component. For the second component, there are 2 nodes that are actually inserted. For primitive insertion, these are the two processes that are needed since there is no schema checking required. Therefore, the total process required following the formal model is $4S + 2U$.

On the other hand, in safe insertion, there are 3 traversal paths both vertically and horizontally in each instance (see Fig. 7). The number of nodes to be inserted remains 2 nodes. However, now we have to perform schema checking as well. Following Fig. 14(a), there are 5 paths to be traversed during this schema checking. Therefore, the total process required following the formal model is $11S + 2U$.

For such a simple example, the searching process for safe insertion is almost three times that of primitive insertion. To avoid this problem, we can implement an additional index in the schema. By adding schema, we do not have to

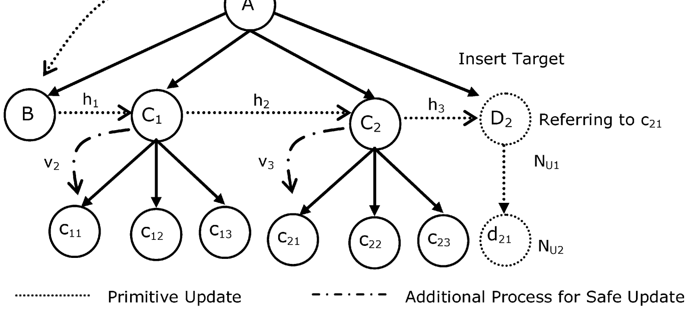


Fig. 7. Referencing insertion in a sample XML document.

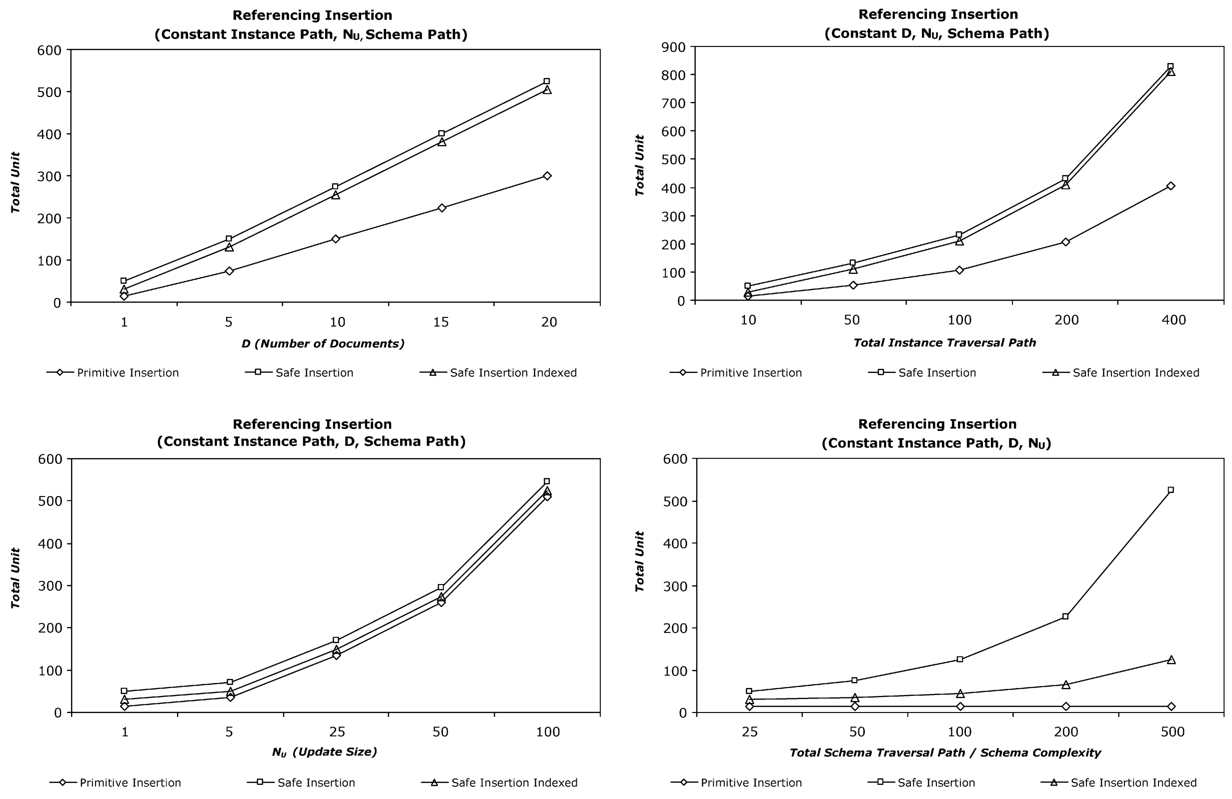


Fig. 8. Performance comparison graphs for referencing insertion.

traverse the whole path in the schema to check all constraints. Say now we implement the index and the size of the schema traversal path can be reduced to 20% of the current size. We call this safe update *safe insertion indexed.*

For this option, the first and second components of the formal model remain the same. However, the schema checking component can be reduced into 1 path. Therefore, the total process required following the formal model is $7S + 2U$.

From the formal model, we draw graphs that can depict the pattern of update complexity with the alternation of parameters. The first parameter that can be altered is the number of documents (D), which is set from 1 to 20 documents. Next is the total search size or traversal path of the document. This parameter is set from 10 to 400. The next parameter is the update size element, which shows the value of N_U. This parameter is set from 1 to 100. The final parameter is the number of paths in the schema that is set from 25 to 500.

The graphs in Fig. 8 show the total processing units for three different insertions. There are two assumptions. First, the instance traversal path for safe update will be twice the instance traversal path for primitive update. This is because in safe update, we have to perform instance checking. Second, we assume that one Update process unit (U) is equivalent to five Search process units (S). This assumption is based on the experimentation of updating one node and searching one node in the same location.

The graphs show that the process units, which represent the computational complexity, grow linearly with the increase of document numbers (D). The process units will grow exponentially with the increase of the total instance traversal paths, number of inserted targets and total schema traversal paths. Notice that by having constraint indexed in the schema, the safe insertion process units can be reduced. To reduce the process units of safe update even more, we can also index the instance document for the future work.

We summarize the rest of insertion comparison for different XML structures in Table 1. Safe insertion with index can reduce the safe update process unit, especially for increasing schema complexity.

Table 1
Insertion performance comparisons for different parameters and constraints

	D					Instance trav. path					N_U					Schema complexity				
	1	5	10	15	20	10	50	100	200	400	1	5	25	50	100	25	50	100	200	500
Dereferencing insertion																				
Primitive	15	75	150	225	300	15	55	105	205	405	15	35	135	260	510	15	15	15	15	15
Safe	50	150	275	400	525	50	130	230	430	830	50	70	170	295	545	50	75	125	225	525
Safe indexed	30	130	255	380	505	30	110	210	410	810	30	50	150	275	525	30	35	45	65	125
Aggregation insertion																				
Primitive	15	75	150	225	300	15	55	105	205	405	15	35	135	260	510	15	15	15	15	15
Safe	50	150	275	400	525	50	130	230	430	830	50	70	170	295	545	50	75	125	225	525
Safe indexed	30	130	255	380	505	30	110	210	410	810	30	50	150	275	525	30	35	45	65	125
Union inheritance sub-type insertion																				
Primitive	5	25	50	75	100	5	5	5	5	5	5	25	125	250	500	5	5	5	5	5
Safe	35	75	125	175	225	45	105	180	330	630	45	65	165	290	540	45	70	120	220	520
Safe indexed	15	55	105	155	205	25	85	160	310	610	25	45	145	270	520	25	30	40	60	120
Mutual exclusion inheritance sub-type insertion																				
Primitive	5	25	50	75	100	5	5	5	5	5	5	25	125	250	500	5	5	5	5	5
Safe	45	125	225	325	425	55	155	280	530	1030	55	75	175	300	550	55	80	130	230	530
Safe indexed	25	105	205	305	405	35	135	260	510	1010	35	55	155	280	530	35	40	50	70	130
Multiple inheritance insertion																				
Primitive	5	25	50	75	100	5	5	5	5	5	5	25	125	250	500	5	5	5	5	5
Safe	45	125	225	325	425	50	130	230	430	830	50	70	170	295	545	50	75	125	225	525
Safe indexed	25	105	205	305	405	30	110	210	410	810	30	50	150	275	525	30	35	45	65	125

5.2. Deletion evaluation

Unlike insertion, for deletion in an association relationship, we are concerned only with the deletion of referred node(s), which we call dereferencing deletion. The deletion of a referencing node(s) will be treated like a deletion of simple value node(s), such as in an aggregation relationship.

The example of dereferencing deletion is if we want to delete a C_2 element and its children nodes (see Fig. 9), including c_{21}. We see that there is a node D_2 that refers to c_{21}. For schema checking, we use the same traversal paths as in Fig. 13(a).

For primitive deletion, we do not need to check whether the target actually has referring node(s). The process of deletion only incurs traversal process to the target location, in this case C_2. The target C_2 consists of 4 nodes, which represents the N_U component in our formal model. Assuming we want to delete C_2 node in one document only, the value of m is equal to 1. The horizontal traversal path is equal to 2 and the vertical traversal path is equal to 2. These two values determine the values of n and p respectively.

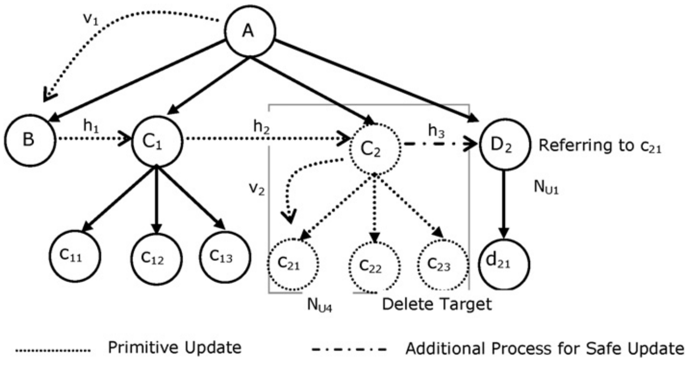


Fig. 9. Dereferencing deletion in a sample XML document.

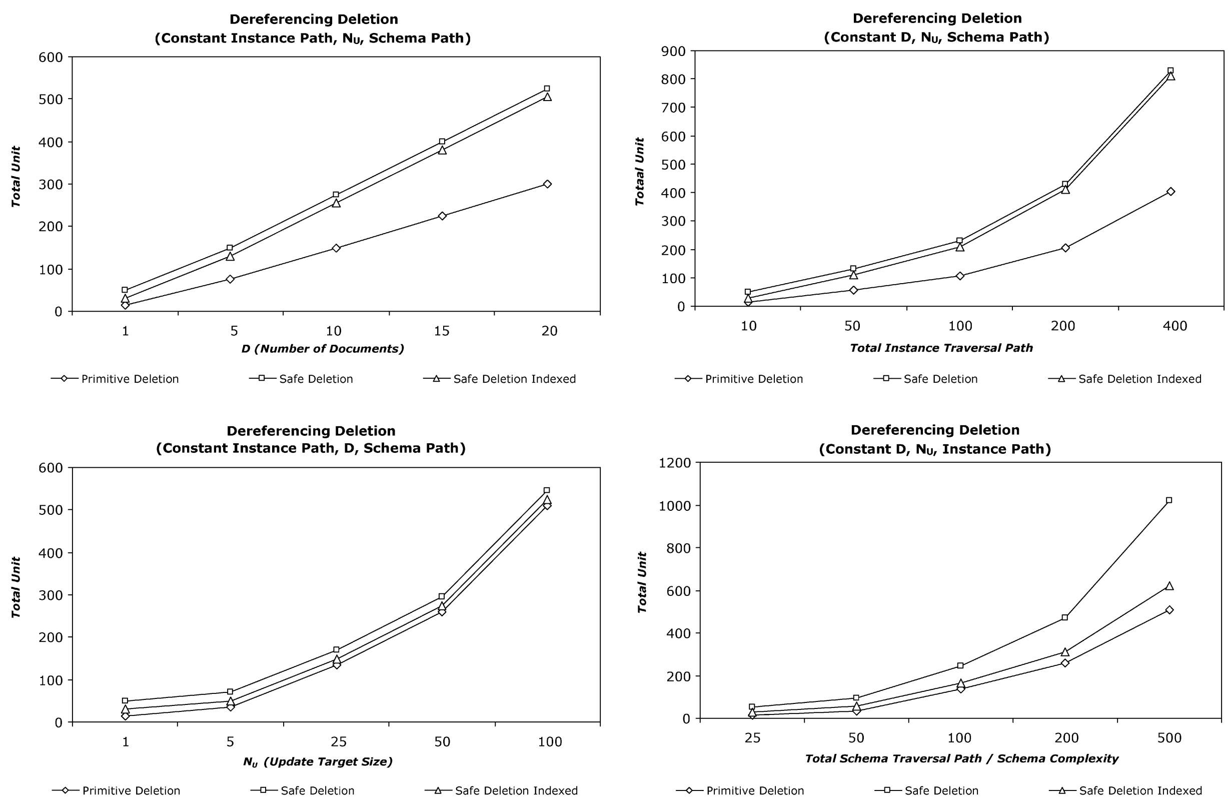


Fig. 10. Performance comparison graph for dereferencing deletion.

For safe deletion, we need to check all nodes that have reference to node C and its elements. In primitive deletion, there are 2 horizontal and vertical paths to search for each instance. Also, there are 4 nodes to be deleted (see Fig. 9). Therefore, the total process unit for primitive deletion is $4S + 4U$.

In safe deletion, we have to traverse 3 horizontal paths and 2 vertical for each instance traversal and 5 paths for schema traversal. The number of deleted node is still 4 nodes. Therefore, the total processing units for safe deletion is $10S + 4U$. Once we perform indexing in the schema, we can reduce the third processing unit and the total processing unit is reduced to $6S + 4U$.

The graphs in Fig. 10 show the total processing units for three different deletions. The assumptions still follow the assumptions from the previous section.

The graphs' tendency for the deletion operation is very similar to those of the insertion operation. The processing units grow linearly with the increase in document number (D). They will grow exponentially with the increase of the total instance traversal paths, number of deleted targets and total schema traversal paths. Notice, that by having constraint indexed in the schema, the safe deletion processing unit can be reduced.

We summarize the rest of the comparison of different XML structures in Table 2. Safe Update with index can reduce the safe update processes, especially for increasing schema complexity.

5.3. Replacement evaluation

In this section we compare the primitive and safe replacement processing units. There are two types of replacement to maintain the constraints in association relationships and two types of replacement for constraints in inheritance relationships. We show the graph comparison in Table 3.

Note that the trend of the replacement for each parameter change is very similar to the trends of previous update operations. The use of index also reduces the safe replacement processing unit even though it will not reach down close to the primitive replacement.

Table 2
Deletion performance comparisons for different parameters and constraints

	D					Instance trav. path					N_U					Schema complexity				
	1	5	10	15	20	10	50	100	200	400	1	5	25	50	100	25	50	100	200	500
Aggregation deletion																				
Primitive	15	75	150	225	300	15	55	105	205	405	15	35	135	260	510	15	35	135	260	510
Safe	50	150	275	400	525	50	130	230	430	830	50	70	170	295	545	50	95	245	470	1020
Safe indexed	30	130	255	380	505	30	110	210	410	810	30	50	150	275	525	30	55	165	310	620
Inheritance sub-type deletion																				
Primitive	15	75	150	225	300	15	75	150	275	500	15	35	135	260	510	15	35	135	260	510
Safe	50	150	275	400	525	50	150	275	500	925	50	70	170	295	545	50	95	245	470	1020
Safe indexed	30	130	255	380	505	30	130	255	480	905	30	50	150	275	525	30	55	165	310	620

5.4. Scalability

It is inevitable that safe update will affect the performance of data manipulation. The computational complexity model also shows this fact, even though it is not as bad as expected. However, with larger data and more constraints to be dealt with, the performance and scalability might deteriorate.

Table 4 summarizes the scalability of the safe update for parameter changes up to 2500%. The value is the average of three different update operations (insertion, deletion and replacement) in three different structures (association, aggregation, inheritance).

From the tables, we can see that the poorest scalability issue occurs in the insertion operation for an inheritance relationship. It is very reasonable since at the moment the super-type and sub-types in inheritance structures have to be implemented as separate documents. Thus, for checking, we require more paths to traverse. Currently, there is no solution to reduce this problem due to the poor representation of inheritance structures in any XML standard.

From the tables, we can also see that the scalability issue occurs if the variable parameters are instance traversal path and schema complexity. To solve this problem, one current work proposes adopting some sort of index mechanism to speed up the checking process. The index has to cater for the dynamic nature of the XML schema as well, since the schema will most likely be dynamic in nature. Once the index has become too big, we can also introduce a filtering mechanism that gives different levels of importance to all constraints. The constraints that have less importance, or that will be less used, will receive the lowest index priority.

Table 3
Replacement performance comparisons for different parameters constraints

	D					Instance trav. path					N_U					Schema complexity				
	1	5	10	15	20	10	50	100	200	400	1	5	25	50	100	25	50	100	200	500
Referencing replacement																				
Primitive	15	75	150	225	300	15	55	105	205	405	15	35	135	260	510	15	15	15	15	15
Safe	50	150	275	400	525	50	130	230	430	830	50	70	170	295	545	50	75	125	225	525
Safe indexed	30	130	255	380	505	30	110	210	410	810	30	50	150	275	525	30	35	45	65	125
Dereferencing replacement																				
Primitive	15	75	150	225	300	15	55	105	205	405	15	35	135	260	510	15	35	135	260	510
Safe	60	200	375	550	725	60	180	330	630	1230	60	80	180	305	555	60	105	255	480	1030
Safe indexed	40	180	355	530	705	40	160	310	610	1210	40	60	160	285	535	40	65	175	320	630
Inheritance sub-type replacement																				
Primitive	15	75	150	225	300	15	55	105	205	405	15	35	135	260	510	15	35	135	260	510
Safe	45	125	225	325	425	45	105	180	330	630	45	65	165	290	540	45	90	240	465	1015
Safe indexed	25	105	205	305	405	25	85	160	310	610	25	45	145	270	520	25	50	160	305	615
Inheritance super-type replacement																				
Primitive	15	75	150	225	300	15	55	105	205	405	15	35	135	260	510	15	35	135	260	510
Safe	50	150	275	400	525	50	130	230	430	830	50	70	170	295	545	50	95	245	470	1020
Safe indexed	30	130	255	380	505	30	110	210	410	810	30	50	150	275	525	30	55	165	310	620

Table 4
Safe update scalability comparisons

	Parameter (increase 25x)			
	D	Instance trav. path	N_U	Schema complexity
Safe update: primitive update				
Association	1.883	2.293	1.274	21.804
Aggregation	1.750	2.098	1.259	18.500
Inheritance	2.611	43.921	1.303	53.498
Safe update indexed: primitive update				
Association	1.817	2.195	1.126	5.490
Aggregation	1.683	2.000	1.111	4.775
Inheritance	2.478	41.876	1.149	13.106

	Parameter (increase 25x)			
	D	Instance trav. path	N_U	Schema complexity
Safe update: primitive update				
Insertion	2.667	44.049	1.310	70.000
Deletion	1.750	2.004	1.259	2.000
Replacement	1.833	2.220	1.269	10.252
Safe update indexed: primitive update				
Insertion	2.533	42.000	1.156	16.667
Deletion	1.683	1.915	1.111	1.216
Replacement	1.767	2.122	1.120	2.998

Using the index, we can reduce the traversal path number, both in instance level and in schema level. The second part of both tables shows the comparison of safe update with index and primitive update processing units after we implement index in the schema level. It can improve the scalability up to 500 per cent. By having index in the instance level, we can improve the performance.

There will be other possible solutions to solve the performance issue. At the end of the day, we will not get the performance of safe update to match the performance of primitive update. Data integrity will incur some costs. However, further research will try to reduce this overhead.

6. Case study

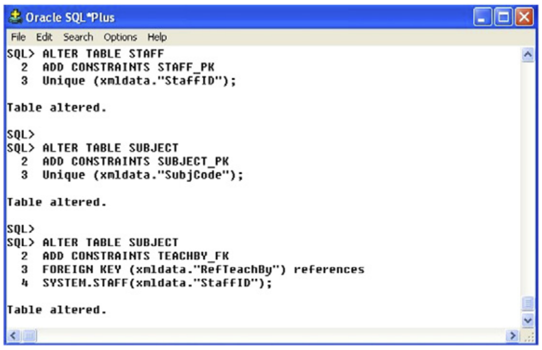
We have proposed the methodologies to preserve XML constraints during the update. Different XEnDB products will implement these methods differently. In this section, we implement our methods using Oracle, one of the XEnDB products that are based on (Object)-Relational Model [36].

Oracle claims it has supported XML storage since its Oracle 8i version. However, not until Oracle 9i Release 2 has it supported the use of XML Type for the XML data. For this implementation, we use Oracle 10g Release 1. At the time of writing, the vendor has launched the Oracle 10g Release 2. The proposed methods are implemented as PL/SQL triggers. Inside these triggers we will use some built-in functions, specifically XML Type [39].

We use the case study *Faculty* and *Staff* documents as they are shown in Fig. 4 and the inheritance relationship under *Staff* document is shown in Fig. 5. The first document is used for methodologies related to association and aggregation relationships and the second document is used for the methodology related to inheritance relationship.

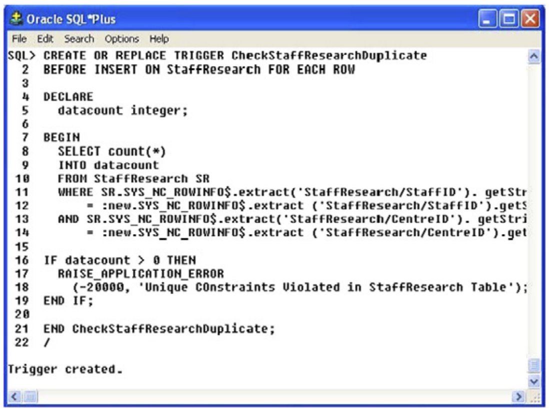
Before we implement the update methodologies, we create the tables to store the XML data. The tables will be based on the schema that has captured all the XML conceptual constraints. Before we use the schema, we need to register it in our database. Once it is registered, we can use all or part of the schema to create a table or a column.

Instead of having one table for the whole document, we create a separate table for each complex type. By doing this, we can utilize more update facilities provided by Oracle. As one example, we cannot perform a subject insertion if we store the XML data as a faculty document in a single table. Another example is that, if we store the XML as a large tree, we cannot insert built-in constraints to elements in a low tree hierarchy. Once the storages are ready, we can start to implement the update methodologies.



```
SQL> ALTER TABLE STAFF
  ADD CONSTRAINTS STAFF_PK
  Unique (xmldata."StaffID");

Table altered.

SQL>
SQL> ALTER TABLE SUBJECT
  ADD CONSTRAINTS SUBJECT_PK
  Unique (xmldata."SubjCode");

Table altered.

SQL>
SQL> ALTER TABLE SUBJECT
  ADD CONSTRAINTS TEACHBY_FK
  FOREIGN KEY (xmldata."RefTeachBy") references
  SYSTEM.STAFF(xmldata."StaffID");

Table altered.
```

(a) Using built-in constraint mechanism.



```
SQL> CREATE OR REPLACE TRIGGER CheckStaffResearchDuplicate
  BEFORE INSERT ON StaffResearch FOR EACH ROW

  DECLARE
    datacount integer;

  BEGIN
    SELECT count(*)
    INTO datacount
    FROM StaffResearch SR
    WHERE SR.SYS_NC_ROWINFO$.extract('StaffResearch/StaffID'). getStr
        = :new.SYS_NC_ROWINFO$.extract ('StaffResearch/StaffID').getS
    AND SR.SYS_NC_ROWINFO$.extract('StaffResearch/CentreID'). getStri
        = :new.SYS_NC_ROWINFO$.extract ('StaffResearch/CentreID').get

  IF datacount > 0 THEN
    RAISE_APPLICATION_ERROR
      (-20000, 'Unique COnstraints Violated in StaffResearch Table');
  END IF;

  END CheckStaffResearchDuplicate;
  /

Trigger created.
```

(b) Using trigger mechanism.

Fig. 11. KEY/KEYREF creation using two mechanisms.

First is for the insertion operation in the association relationship. Algorithms 1 and 2 show generic methodologies that should be followed by any XEnDB product. As in this implementation, all attributes and elements are implemented equally as table attributes; we do not differentiate the implementation for these two algorithms.

Since Oracle 10g is based on an object-relational data model, we should be able to implement the insertion methodology (Algorithm 1) by using primary and foreign key mechanisms. The primary and foreign keys are the physical representation of logical KEY/KEYREF in XML Schema. Unfortunately, at the time of writing, the KEY/KEYREF in XEnDB Schema is not supported by Oracle 10g. When we map the schema into object-relational tables, the information regarding the KEY and KEYREF is abandoned.

Fortunately, we can preserve the KEY/KEYREF constraints after the table is created by table alteration. However, they are applicable only for the table attribute of simple data type (see Fig. 11(a)). We cannot add constraints to the attribute of user defined type or array. A problem arises when we want to add the unique constraints to composite attributes. In the schema, we have element *StaffResearch*, which is derived from the relationship between *Staff* and *ResearchCentre* elements. For this special case we need to use a trigger to enforce the unique constraints (see Fig. 11(b)).

The unique constraint enables us to preserve the key semantic in the XML document. Now we cannot insert a duplicate key VALUE in the database. Also, we cannot insert a KEYREF value if the intended KEY does not exist in the first place. In this case, we follow the restrict maintenance strategy. Figure 12 shows how these invalid insertions raised an error during execution.

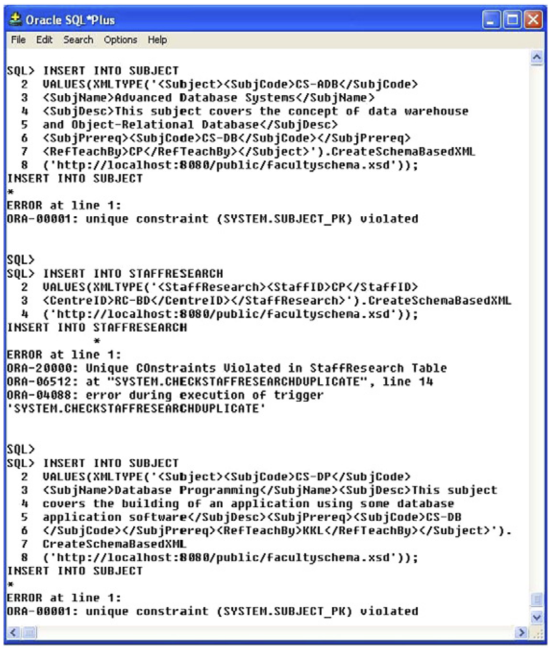
Similar to insertion, we can control the deletion and replacement of KEY/KEYREF both by having the constraints defined in the table and by using the trigger.

The checking of the other constraints such as cardinality, adhesion, ordering, etc., is a lot simpler. Oracle has provided some built-in functions that check the document being updated with its schema.We just need to contain the built-in function into a trigger and set it to fire every time we need it. In Fig. 13(a) we create triggers that will be fired before we delete a data in *Office* table. Inside the triggers, we use the function *schemaValidate*( ). Note that, since in a table we cannot delete an attribute, we have to use replace/update instead of delete. Figure 13(b) shows how the trigger is executed.

These few samples show how we can maintain the XML constraints after update operations. The proposed methodologies can be implemented using user-defined functions or using an existing constraint mechanism provided by the DBMS.

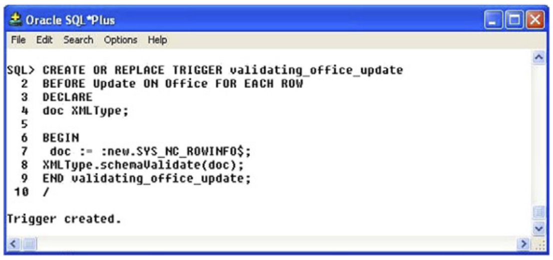
7. Conclusion and future work

XEnDB, which has emerged as a combination of traditional DBMS and Native XML Database, is a popular solution for XML data repository. However, not many works have been done to research this database family. Our work investigates the aspect of updating XML documents stored in XEnDB. The goal is to propose methodologies that maintain the semantic of the documents after some update operations.



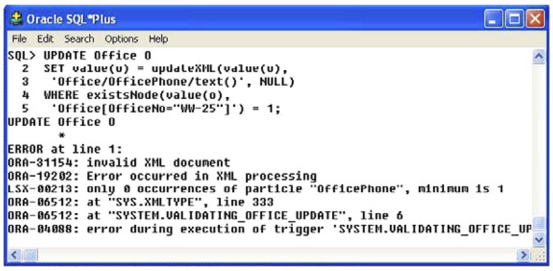
```
SQL> INSERT INTO SUBJECT
  2  VALUES(XMLTYPE('<Subject><SubjCode>CS-ADB</SubjCode>
  3  <SubjName>Advanced Database Systems</SubjName>
  4  <SubjDesc>This subject covers the concept of data warehouse
  5  and Object-Relational Database</SubjDesc>
  6  <SubjPrereq><SubjCode>CS-DB</SubjCode></SubjPrereq>
  7  <RefTeachBy>CP</RefTeachBy></Subject>').CreateSchemaBasedXML
  8  ('http://localhost:8080/public/facultyschema.xsd'));
INSERT INTO SUBJECT
*
ERROR at line 1:
ORA-00001: unique constraint (SYSTEM.SUBJECT_PK) violated


SQL>
SQL> INSERT INTO STAFFRESEARCH
  2  VALUES(XMLTYPE('<StaffResearch><StaffID>CP</StaffID>
  3  <CentreID>RC-BD</CentreID></StaffResearch>').CreateSchemaBasedXML
  4  ('http://localhost:8080/public/facultyschema.xsd'));
INSERT INTO STAFFRESEARCH
            *
ERROR at line 1:
ORA-20000: Unique COnstraints Violated in StaffResearch Table
ORA-06512: at "SYSTEM.CHECKSTAFFRESEARCHDUPLICATE", line 14
ORA-04088: error during execution of trigger
'SYSTEM.CHECKSTAFFRESEARCHDUPLICATE'


SQL>
SQL> INSERT INTO SUBJECT
  2  VALUES(XMLTYPE('<Subject><SubjCode>CS-DP</SubjCode>
  3  <SubjName>Database Programming</SubjName><SubjDesc>This subject
  4  covers the building of an application using some database
  5  application software</SubjDesc><SubjPrereq><SubjCode>CS-DB
  6  </SubjCode></SubjPrereq><RefTeachBy>KKL</RefTeachBy></Subject>').
  7  CreateSchemaBasedXML
  8  ('http://localhost:8080/public/facultyschema.xsd'));
INSERT INTO SUBJECT
*
ERROR at line 1:
ORA-00001: unique constraint (SYSTEM.SUBJECT_PK) violated
```

Fig. 12. Algorithm 1 implementation examples through keys & triggers.



```
SQL> CREATE OR REPLACE TRIGGER validating_office_update
  2  BEFORE Update ON Office FOR EACH ROW
  3  DECLARE
  4  doc XMLType;
  5
  6  BEGIN
  7   doc := :new.SYS_NC_ROWINFO$;
  8  XMLType.schemaValidate(doc);
  9  END validating_office_update;
 10  /

Trigger created.
```

(a) Aggregation deletion using trigger mechanism.



```
SQL> UPDATE Office O
  2  SET value(o) = updateXML(value(o),
  3   'Office/OfficePhone/text()', NULL)
  4  WHERE existsNode(value(o),
  5   'Office[OfficeNo="WW-25"]') = 1;
UPDATE Office O
       *
ERROR at line 1:
ORA-31154: invalid XML document
ORA-19202: Error occurred in XML processing
LSX-00213: only 0 occurrences of particle "OfficePhone", minimum is 1
ORA-06512: at "SYS.XMLTYPE", line 333
ORA-06512: at "SYSTEM.VALIDATING_OFFICE_UPDATE", line 6
ORA-04088: error during execution of trigger 'SYSTEM.VALIDATING_OFFICE_UP
```

(b) Aggregation deletion trigger checking.

Fig. 13. Deletion methodology implementation (aggregation relationship) and checking.

To achieve our target, we firstly identify different XML conceptual constraints. Then, we transform the conceptual constraints into the logical model constraints. In this case, we elect to use SQL/XML Schema as the logical model representation.

We use the schema to propose the generic update methodologies. We differentiate the methods based on the operation types: insertion, deletion and replacement. We also consider different updated targets and different maintenance strategies. These generic methods are the basic functions that need to be followed by any XML storage regardless of the underlying data model.

We have also implemented these generic methods by using one of the most widely-used XEnDBs, Oracle 10g. By using the schema, we map complex elements into XML Type. Each XML Type is then mapped into separate tables. We do not implement a whole document into one table for two reasons. First, we cannot embed many Oracle built-in constraint mechanisms in a deep tree structure. Therefore, by keeping them separate, we avoid a high volume of user-defined functions, which can be costly. Second, there is major limitation in data manipulation if they are stored as an XML Type. Every time we want to manipulate a single element, we have to work on the whole document. It is of course very costly and is at odds with the purpose of our work.

Once the data has been stored, we implement the update methodologies using Oracle built-in constraints and some user-defined triggers written in PL/SQL. For some triggers, we need a duplicate table to enable the checking. This is one shortcoming of this current work, since it is certainly costly to have duplicate values and duplicate operations.

Nevertheless, to maintain the integrity of the data, some overhead cost is inevitable. In addition, this is applicable only to tables that have a high frequency of updates.

Most of the constraints identified in the conceptual model can be captured using Oracle 10g. The only limitation is that it does not differentiate an attribute from an element. Therefore, during the mapping process, we map the attribute as an element with simple type. The built-in function that checks the XML instance with a registered schema also simplifies many of the checking mechanisms. With some other products, a more user-defined function might be needed.

We also evaluate the proposed update methodologies for safely updating XML documents. We use a formal technique that models the update processes. With this model, the evaluation is independent of the database model or products.

It is obvious that the processing units of update with the proposed methodologies will be higher than those of primitive update. This is because the new update methodologies are concerned with the consistency of the XML constraints. Therefore, every time an update is performed, the system will check the XML schema and the XML instance beforehand. If the operation does not violate the original constraints, then it can proceed. There are four determining parameters for update processing units: the number of documents, the size of the update target, the instance traversal path and the schema complexity. By alternating these parameters, we can analyse the pattern of different update operations.

Despite our attempt to undertake a thorough research on safe update methodologies in XML-Enabled Database, some issues have arisen from this work. These issues can be used as further research topics in XML data management.

Scalability of safe update is an important issue. Even though we propose to use an index to reduce the search traversal path, in some XML relationships the scalability is still poor. Further work is necessary in this area. Another important further research area is the possibility of constraint changes. The current work covers only the static constraints. Therefore, if there are some changes in the constraints, the users have to redo the mappings and redo the implementation functions/triggers. Methodologies to capture the changes of schema can be proposed and these methodologies can be used to automatically alter the mapping/transformation process and alter the implementation functions/trigger.

As a final note, XML update with constraints checking is not desirable because of the considerably high cost. However, due to the increased use of XML database, the need to have consistent data is inevitable. The constraint preservation is a necessity, not simply an option. In addition, in some cases, the proposed update methodologies do not significantly degrade the performance of update operations.

Our new update methodologies will derive significant benefit from continuous research on hardware and XML storage options. This research will hopefully produce methodologies which reduce both search and update costs.

References

[1] P. Amornsinlaphachai, M.A. Ali, B.N. Rossiter, Updating XML using object-relational database, in: Lecture Notes in Comput. Sci., vol. 3567, Springer, 2005, pp. 155–160.
[2] Apache Software Foundation: Apache Xindice, http://xml.apache.org/xindice/, 2005.
[3] R. Bourett, XML and databases, http://www.rpbourret.com/xml/XMLAndDatabases.htm, last updated December 2004.
[4] R. Bourett, XML database products, http://www.rpbourret.com/xml/XMLDatabaseProds.htm, last updated January 2005.
[5] R. Bourett, XML database products: XML-enabled databases, http://www.rpbourret.com/xml/ProdsXMLEnabled.htm, last updated January 2005.
[6] P. Buneman, W. Fan, J. Siméon, S. Weinstein, Constraints for semi-structured data and XML, SIGMOD Record 30 (1) (2001) 45–47, ACM Press.
[7] DB2 text extender, http://www-306.ibm.com/software/data/db2/extenders/text/, April 2005.
[8] DB2 XML extender, http://www-306.ibm.com/software/data/db2/extenders/xmlext/, April 2005.
[9] dbXML Group: dbXML (native XML database), http://www.dbxml.com/product.html, 2005.
[10] D. DeHaan, D. Toman, M.P. Consens, M.T. Özsu, A comprehensive XQuery to SQL translation using dynamic interval encoding, in: SIGMOD 2003, ACM Press, 2003, pp. 623–634.
[11] F. Du, S. Amer-Yahia, J. Freire, ShreX: Managing XML documents in relational databases, in: VLDB 2004, Morgan Kaufmann, 2004, pp. 1297–1300.
[12] A. Eisenberg, J. Melton, SQL/XML is making good progress, SIGMOD Record 30 (2) (2002) 101–108.
[13] R. Elmasri, S. Navathe, Fundamentals of Database Systems, Addison–Wesley, 2003.
[14] W. Fan, XML constraints: Specification, analysis, and applications, in: DEXA Workshops 2005, IEEE CS, 2005, pp. 805–809.
[15] P. Fortier, SQL3 Implementing the SQL Foundation Standard, McGraw–Hill, 1999.

[16] L. Fegaras, R. Elmasri, Query engines for web-accessible XML data, VLDB J. (2001) 251–260.
[17] D. Florescu, D. Kossmann, Storing and querying XML data using an RDMBS, IEEE Data Eng. Bull. 22 (3) (1999) 27–34.
[18] R. Goldman, J. McHugh, J. Widom, From semistructured data to XML: Migrating the Lore data model and query language, in: WebDB, INRIA, 1999, pp. 25–30.
[19] W.-S. Han, K.-H. Lee, B.S. Lee, An XML storage system for object-oriented/object-relational DBMSs, J. Object Technol. 2 (1) (2003) 113–126, ETH Swiss Federal Institute of Technology.
[20] ISO/IEC, Information technology, database languages, SQL, Part 14: XML-related specifications (SQL/XML), ISO/IEC 9075-14, 2003.
[21] H.V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakhsmanan, A. Nierman, S. Paprizos, J.M. Patel, D. Srivastava, N. Wiwattana, Y. Wu, C. Yu, TIMBER: A native XML database, VLDB J. 11 (4) (2002) 279–291.
[22] B. Kane, H. Su, E.A. Rundensteiner, Consistently updating XML documents using incremental constraint check queries, in: WIDM 2002, ACM Press, 2002, pp. 1–8.
[23] L. Khan, Y. Rao, A performance evaluation of storing XML data in relational database management systems, in: WIDM 2001, ACM Press, 2001, pp. 31–38.
[24] M. Klettke, H. Meyer, XML and object-relational database systems—Enhancing structural mappings based on statistics, in: Lecture Notes in Comput. Sci., vol. 1997, Springer, 2001, pp. 151–170.
[25] W.M. Meier, eXist Native XML Database, in: A.B. Chauduri, A. Rawais, R. Zicari (Eds.), XML Data Management: Native XML and XML-Enabled Database System, Addison–Wesley, 2003, pp. 43–68.
[26] T. Lahiri, S. Abiteboul, J. Widom, Ozone: Integrating structured and semistructured data, in: Lecture Notes in Comput. Sci., vol. 1949, 2000, pp. 297–323.
[27] P. Larson, XML data management go native or spruce up relational systems?, in: SIGMOD 2001, ACM Press, 2001, p. 620.
[28] P. Lehti, Design and implementation of a data manipulation processor for an XML query language, Technische Universität Darmstadt, 2001.
[29] L. Martin, XML update language requirements, http://xmldb-org.sourceforge.net/xupdate/xupdate-req.html, November 2000, XML DB Working Draft.
[30] W.M. Meier, eXist native XML database, in: A.B. Chauduri, A. Rawais, R. Zicari (Eds.), XML Data Management: Native XML and XML-Enabled Database System, Addison–Wesley, 2003, pp. 43–68.
[31] W.M. Meier, eXist: An open source native XML database, in: Lecture Notes in Comput. Sci., vol. 2593, Springer, 2002, pp. 169–183.
[32] S.K. Mostéfaoui, M. Younas, Context-oriented and transaction-based service provisioning, Int. J. Web Grid Services 2 (2007), in press, Inderscience Press.
[33] D. Obasanjo, On the death of the XML database, available online, March 2003.
[34] E. Pardede, J.W. Rahayu, D. Taniar, Preserving conceptual constraints during XML updates, Int. J. Web Inform. Systems (IJWIS) 1 (2) (June 2005) 65–82, Troubador Publishing.
[35] E. Pardede, J.W. Rahayu, D. Taniar, Object-relational complex structures for XML storage, Inform. Softw. Technol. J. (IST) 48 (6) (June 2006) 370–384, Elsevier Science.
[36] E. Pardede, J.W. Rahayu, D. Taniar, XML-enabled relational database for XML document update, in: AINA 2006, WAMIS Workshop, IEEE Computer Society, 2006, pp. 205–211.
[37] J.W. Rahayu, D. Taniar, E. Pardede, Object-Oriented Oracle, IRM Press, 2006.
[38] J. Robie, XQuery: A guided tour, in: H. Kattz (Ed.), XQuery from the Experts, Addison–Wesley, 2004, pp. 3–78.
[39] L.I. Rusu, J.W. Rahayu, D. Taniar, A methodology for building XML data warehouses, Int. J. Data Warehous. Mining (IJDWM) 1 (2) (2005) 23–48, Idea Group Inc.
[40] M. Scardina, B. Chang, J. Wang, Oracle Database 10g XML & SQL: Design, Build & Manage XML Applications in Java, C, C++, & PL/SQL, McGraw–Hill, Osborne, 2004.
[41] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, J.F. Naughton, Relational databases for querying XML documents: Limitations and opportunities, in: VLDB 1999, Morgan Kaufmann, 2002, pp. 302–314.
[42] T. Shimura, M. Yoshikawa, S. Uemura, Storage and retrieval of XML documents using object-relational databases, in: Lecture Notes in Comput. Sci., vol. 1677, 1999, pp. 206–217.
[43] Sleepy Cat: Berkeley DB XML, http://www.sleepycat.com/products/bdbxml.html, 2006.
[44] M. Stonebraker, D. Moore, Object-Relational DBMSs: The Next Great Wave, Morgan Kaufmann, 1996.
[45] I. Tatarinov, Z.G. Ives, A.Y. Halevy, D.S. Weld, Updating XML, in: SIGMOD 2001, ACM Press, 2001, pp. 413–424.
[46] W3C, S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon (Eds.), XQuery 1.0: An XML Query Language, W3C Candidate Recommendation, 3 November 2005, http://www.w3.org/TR/xquery, 2005.
[47] W3C, D. Chamberlin, J. Robie (Eds.), XQuery Update Facility Requirements, W3C Working Draft, 3 June 2005, http://www.w3.org/TR/xquery-update-requirements/, 2005.
[48] N.D. Widjaya, D. Taniar, J.W. Rahayu, Transformation of XML schema to object-relational database, in: D. Taniar, J.W. Rahayu (Eds.), Web Information Systems, Idea Group Publishing, 2004, pp. 141–189.
[49] R.W. Wong, J.C. Curran, TXP: A transaction-based XML parser, in: APWEB 1999, IEEE, 1999.