Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information and Software Technology 48 (2006) 370–384

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Object-relational complex structures for XML storage

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] School of Business Systems, Monash University, Clayton VIC 3800 Australia

Received 31 October 2005; accepted 2 December 2005
Available online 8 February 2006

Abstract

XML data can be stored in various database repositories, including Object-Relational Database (ORDB). Using an ORDB, we get the benefit of the relational maturity and the richness of Object-Oriented modeling, including various complex data types. These data types resemble the true nature of XML data and therefore, the conceptual semantic of XML data can be preserved. However, very often when the data is stored in an ORDB repository, they are treated as purely flat tables. Not only do we not fully utilize the facilities in current ORDB, but also we do not preserve the conceptual semantic of the XML data.

In this paper, we propose novel methodologies to store XML data into new ORDB data structures, such as user-defined type, row type and collection type. Our methodology has preserved the conceptual relationship structure in the XML data, including aggregation, composition and association. For XML data retrieval, we also propose query classification based on the current SQL.

Compared to the existing techniques, this work has several contributions. Firstly, it utilizes the newest features of ORDB for storing XML data. Secondly, it covers a full database design process, from the conceptual to the implementation phase. Finally, the proposed transformation methodologies maintain the conceptual semantics of the XML data by keeping the structure of the data in different ORDB complex structures.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Object Relational Database; User-Defined Type; Row Type; Collection Type; XML; XML Schema

1. Introduction

Object-Relational Database (ORDB) is increasingly popular as a database storage for XML data [17,24]. Its popularity is associated with its ability to capture the object-oriented modeling semantic and the maturity of relational implementation. With many new data structures introduced in the current SQL [7,9,16,18], the modeling power of ORDB has gradually increased. In particular, complex data structures such as row type, user-defined type and collection types can be utilized in XML storage.

There are some works that have discussed the usage of ORDB for XML storage. They map different schema languages such as DTD and XML Schema [1] into the Object-Relational (OR) Schema. However, very often when the data is stored in a database repository, some complex structures are flattened [2,8,10,14,26,31]. This implementation does not utilize ORDB full capabilities. In addition, it has ignored the semantic constraints of the data. We should preserve the semantic constraints in the logical and implementation level for a better solution [21].

This paper proposes models that can preserve the complex structures in XML into ORDB. We perform two mapping steps. First is the mapping from the conceptual model using Semantic Network Diagram [6] to the logical model using XML Schema [28]. Second, the logical model is mapped into a physical implementation using SQL in ORDB. For this purpose, we use the current complex structures in SQL4. We need to emphasize that this work will not cover XML-Enabled features in ORDB, such as LOB and XML Type since, they have not been implemented uniformly by ORDB products.

The paper is structured as follows. Section 2 briefly introduces a number of new ORDB data types that have been standardized in SQL. We only include the types that will be used for XML data storage. Section 3 discusses the background of our work and the existing work in this area. In Section 4, we propose our transformation methodologies that are classified based on the relationship structure. Section 5 describes the queries to retrieve data stored in ORDB using complex structures. We will compare our query performance with other available techniques in Section 6. Finally, we will conclude our work in Section 7.

* Corresponding author. Tel.: +61 3 9479 1280; fax: +61 3 9479 3060.
E-mail addresses: E.Pardede@latrobe.edu.au, wenny@cs.latrobe.edu.au (J.W. Rahayu), david.taniar@infotech.monash.edu.au (D. Taniar).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2005.12.015

2. ORDB new data structures: a preliminary

New data structures in ORDB enable database designers to represent XML data as they are described in the conceptual model [25]. In this section, we introduce three data types that will be used in the subsequent sections.

2.1. User-defined type

User-Defined Type (UDT) in ORDB resembles a class in an Object-Oriented Database. It comprises a number of attributes and routines. The routines support the storage and the manipulation of the complex structure. The internal structure is encapsulated. Access to instances or attributes is done through their routines. In this work, we emphasize the static aspect of UDT only. General syntax for the SQL4 UDT is shown as follows. UDT may exist as *value UDT* or *object UDT*. The former means that the type is bound to a table as column object. The latter on the other hand is shown in an object table as special kind of table.

```
SQL UDT Syntax
  CREATE [OR REPLACE] TYPE <object schema 1> AS OBJECT
     (att1      attribute type,....,
      att1+n    attribute type,

      MEMBER PROCEDURE <procedure name>
         [(parameter [{IN | OUT | IN OUT}] parameter type,....,
           parameter [{IN | OUT | IN OUT}] parameter type)],
        BEGIN
           <procedure body>;
        END <member procedure name>;
      );
 /
```

2.2. Row type

Fortier and Melton [9,16] define a row type as a constructed type that contains a sequence of attribute names and their data types. This type is actually not a new data type in the database systems. It has been used even since the legacy data model era [4].

After the emergence of relational model, there is also a data model that is aimed at capturing the nested structure such as row type in relations. The model is called Nested Relational Model (NRM) [12,23]. Nevertheless, the traditional relational model still dominates the database community. Even until recently, there is no commercial DBMS which has chosen to implement the NRM even in its original form [5].

Not until the release of SQL3, did the Relational Model recognizes Row Type as one data type that can enrich its data structure [9]. In SQL4, it is even possible to have varying levels of row type [16]. It is very useful to model a real world problem that can rarely be represented by a simple flat table. The general syntax for the SQL4 row type is shown in the following code.

```
SQL4 Row Type
   CREATE TABLE <table schema>
      (attr1 data type
       CONSTRAINT attr1 PRIMARY KEY, ....,
       attri ROW (attri1 data type,...,
                  attrij data type));
```

2.3. Collection type

Collection types are formulated by Object Database Management Group (ODMG) [3]. A collection is composed of distinct elements, each of which can be of a simple data type, constructed data type or UDT. An important distinguishing characteristic is that all collection elements must be of the same type.

ODMG has defined four collection types: *sets*, *lists*, *arrays*, and *bag* (*multiset*) [3]. However, at the time of writing, SQL4 [9,16] has only standardized array and multiset. An array is a dynamically sized ordered collection that allows duplicates. A multiset is an unordered collection that allows duplicates. The collection type can be constructed by simple data types (such as INTEGER), constructed data types (such as ROW), or UDT.

```
SQL Collection Syntax
   CREATE TABLE <table schema>
      (att1 data_type CONSTRAINT attr1 PRIMARY KEY,...,
       att1+n ARRAY[NUMBER] (<simple|constructed|UDT>));
```

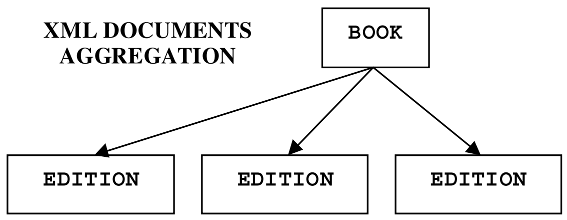
For simple data types, we can use any predefined types such as integer, char, etc. For constructed data types we will use the ROW type, which contains a sequence of attribute names and data types [9,16]. Finally, we will also use UDT inside a collection as to accommodate the sharing constraints in an aggregation relationship.

3. Background and related works

In the last few years, we have witnessed many works that aim to store the XML data into different kind of repositories. A few researchers and products have chosen a novel way by proposing native XML databases that keep the XML in their natural tree-form [11,13,15,27]. Many other researchers believe in the more-established databases and try to tailor the existing technology to suit the XML data requirements. XML storage based on the relational model has proven very popular, especially for data-centric XML document [2]. There is considerable amount of research that works on the usage of relational database (RDB) and object-relational database (ORDB) for XML repository.

ORDB is a powerful database with a rich modeling capability. The solid relational model foundation is enriched with some OO and NRM features. From the OO concept, ORDB adopts TYPE and from the NRM, ORDB adopts the concept of ROW [10]. From both OO and NRM, ORDB adopts the concept of collection types. These complex structures have been implemented by many ORDB vendors for years and have become a standard in SQL3 and the forthcoming SQL4 [18].

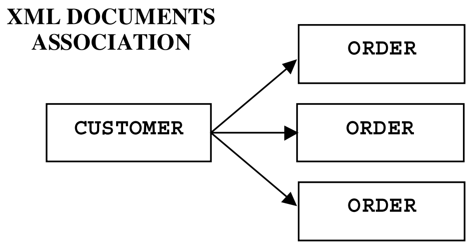
These new data structures open many possibilities of XML data implementation in an ORDB, especially since the complex structure in some ways resemble the XML structure. Some of the RDB limitations listed in [26], such as, support for sets and multiple queries optimization have now been answered. Some conceptual XML structures such as aggregation, composition and association can also be preserved using these new data



```
<xsd:element name = "BOOK" type = "BookType"/>
<xsd:complexType name = "BookType">...
    <xsd:element name = "EDITION" minOccurs =
    ?0? maxOccurs= ?unbounded?/>...
</xsd:complexType>
</xsd:element>
```

Book	
PK	
book_id	book_title
EN1	Fundamentals of Database System

Book-Edition		
PK, FK	PK	
book_id	ed-no	ed-year
EN1	1	1989
EN1	2	1997
EN1	3	2000



```
<xs:element name = "CUSTOMER" type = "CustomerType" / >
<xs:complexType name = "CustomerType">
    <xs:element name = "ORDER" type= "OrderType" max
Occurs = "unbounded"/>...
</xs:complexType>

<xs:complexType name = "OrderType">...
</xs:complexType>
```

Customer	
PK	
CustomerID	CustomerName
MR2	Mark Roberts

Order		
PK	PK	FK
OrderNo	OrderDate	CustomerID
1	25/08/2003	MR2
2	26/08/2003	MR2
3	27/08/2003	MR2

Fig. 1. ORDB flat implementation for XML data.

types. Unfortunately, very often when ORDB is used for XML repository, the complex structures are not utilized. The collection is usually flattened or split into an entirely separate table (see Fig. 1).

Florescu and Kossmann [8] presents a simple mapping of XML data into relational tables. In this work, they treat XML documents as graphs with edges and leaves. The edges represent the relationships while the leaves represent the values. In this work, the aggregation relationship is mapped into separate flat tables by using composite keys. It has diminished the collection semantics in aggregation relationship.

Bourret, Shanmugasundaram and co-workers [2,26] proposes comprehensive mapping from DTD into an OO Schema and the implementation of the results into tables. In the implementation stage, again the aggregation type is flattened. The association relationship is mapped using IDREF. This is usually done when it is not possible to form collection or nesting.

Han et al. [10] proposes mapping from the XML Schema into the OO/OR Schema. The work compares how the mapping into relational schema can be changed into the OR schema. Thus, it does not cover the unique properties of an OR model such as different types of relationships including aggregation. Regarding association relationship, this work has mentioned the usage of collection to store the reference. Nevertheless, it does not show the step-by step mapping from conceptual level down to the implementation.

Klettke and Meyer [14] proposes mapping from the DTD into the OR Schema. The main contribution of the work is the usage of a hybrid database where the users can select certain attributes to be stored in ORDB and others to be stored as they are (as XML data). It does not show the mapping for different kinds of relationship and how we utilize the new data structures in ORDB.

Xiaou et al. [31] proposes the mapping of the OO Conceptual Model into the XML Schema. This work has included collection for aggregation relationship. However, this work does not discuss the usage of collection in composition and association relationship. In addition, it only covers the mapping into logical design. No implementation in OROB is provided.

Finally, Widjaya and co-workers [29,30] propose the mapping of different relationships of XML Schema to ORDB. In [29], the collection in XML Schema disappears in the tables since they store the reference in the 'many' side or in the separate table. In [30], the collection is preserved in the XML Schema, but again the data is stored separately in cluster tables or nested tables.

We find that the existing works have not utilized the full complex structures available in ORDB for their XML repository. By not doing so, their relationship structures semantic are not represented well. In this work, we propose the usage of new ORDB data structures to preserve the semantics of different relationship structures. Our proposed method will cover the transformation from the conceptual to

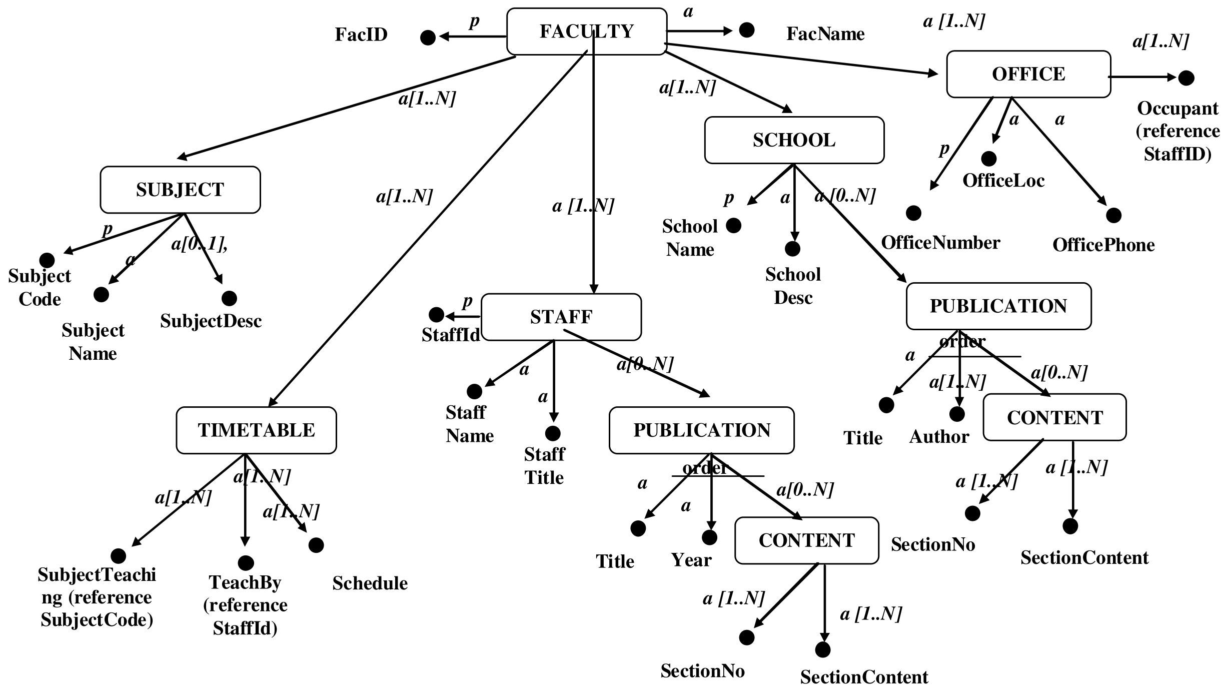


Fig. 2. Faculty XML document diagram.

the implementation level of the database design. In addition, we will also cover the queries to manipulate the data and compare the query performance with other methodologies.

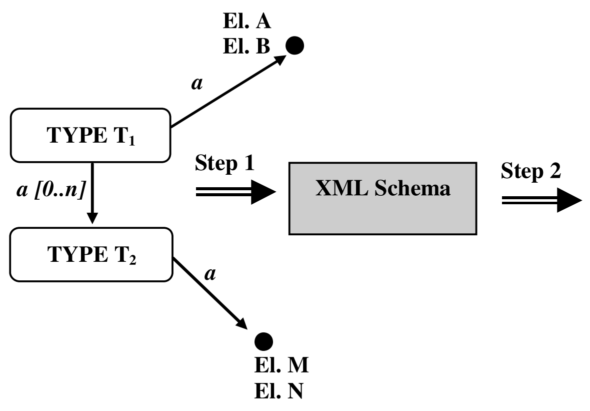
4. Proposed transformation methodology

In this section, we will show the proposed transformation methodology. We will differentiate the mapping based on the relationship structure, where the complex structures can be used.

As for running example, we will use the XML data containing information on a Faculty (see Fig. 2). The diagram used is a Semantic Network Diagram [6] that shows not only complex and simple elements, but also the relationship type and its semantic constraints.

In this diagram, a node can be represented as a box or a small black circle. We use boxes for complex types and small black circles for leaf elements and attributes. The difference between leaf element and attributes are indicated by the lines. If the line connecting the small black circle and the box has the letter ‘a’ (for aggregation), the circle is a leaf element. Otherwise, if the letter is ‘p’ (for property), the circle represents an attribute.

In the following document, the *faculty* offers many *subjects*, has many *timetables*, employs many *staff*, comprises many *schools* and has many *offices*. Each complex type has simple attributes associated with it. For example, a *subject* will have simple attributes *subject code*, *subject name* and *subject description*. Association relationship between complex types is shown by reference attributes. For example, a *timetable* complex type has two reference attributes. The first is *subject teaching* which refer to a *subject code*. The second is *teach by*



TYPE/TABLE T_1			
Att A	Att B	Coll.T_2	
		Att M	Att N
Val A	Val B	Val M_1	Val N_1
		Val M_2	Val N_2

```
General Syntax
CREATE TYPE <T₂ type schema>
(attᵢ    data_type,...,
attᵢ₊ₘ   data_type);

CREATE TYPE|TABLE <T₁ table schema>
(attⱼ    data_type, ....,
 attⱼ₊ₙ  <MULTISET|ARRAY> (<T₂ type schema>));
```

Fig. 3. Aggregation transformation methodology.

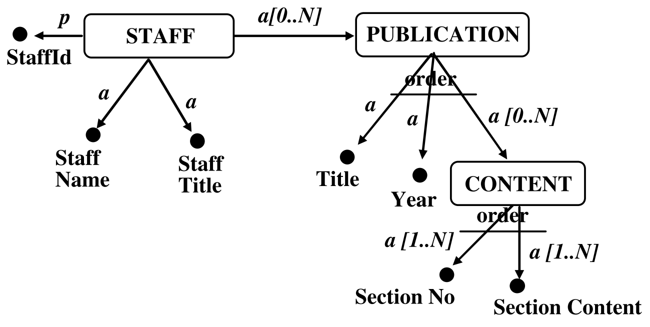


Fig. 4. STAFF aggregation example.

that refers to a *staff ID*. We will use a subset of this example for each proposed methodologies in this section.

4.1. Proposed method for aggregation relationship

Aggregation is a relationship type in which a composite object (whole) consists of component objects (parts). The 'part' components are shareable and their existence is dependent on the 'whole' component. Therefore, we have to enable access to the 'part' component without first accessing the 'whole' component.

In the first step (see Fig. 3), we map both the 'whole' and the 'part' components as complex types. For the second step, we directly map the 'whole' complex type as the table or type and the 'part' complex types as the complex attributes. If there is only one 'part' component for each 'whole' component, we can use single attribute with complex type. Otherwise, we use collection of attributes. The mapping rules are shown below [19].

4.1.1. Rule 1

Step 1: For two types namely T_1 and T_2 with elements/attributes (A,B) and (M,N), respectively, if T_1 can be composed by shareable and existence-independent T_2, implement T_1 as a complex type and T_2 as another complex type accessed as an element in T_1 with maxoccurs constraint set to unbound for cardinality more than one.

Step 2 For two complex types namely T_1 (A,B) and T_2 (M,N), if T_1 can be composed by more than one shareable and existence-independent T_2 implement T_2 as UDT attribute of table or another type T_1. Transformation result is Type|Table $T_1(A, B, \mathrm{UDT}_0^n\, T_2(M, N))$.

4.1.2. Example 1

From the running example (see Fig. 1), we use the example of aggregation between *staff* and *publication* (see Fig. 4). The *publication* can still exist outside *staff* type. For example, it can also appear under *school* type. The aggregation type will be mapped into single or collections of UDT in ORDB table. At this stage, we will treat element *content* as a simple element. In Section 4.2, we will demonstrate the rule for composition relationship. Note that the horizontal line determines the ordering semantic.

In our first transformation, we come up with XML Schema where both the 'whole' and the 'part' components are defined as complex types. Inside the 'whole' complex type, we will

```
Transformation into XML Schema (Step 1):

<xsd:complexType name = "PUBLICATION_Type">
  <xsd:sequence>
    <xsd:element name = "Title" type = "xsd:string"/>
    <xsd:element name = "Year" type = "xsd:integer"/ >
    <xsd:element name = "Content" type = "xsd:string"/ >
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name = "STAFF_Type">
  <xsd:element name = "StaffName" type = "xsd:string"/>
  <xsd:element name = "StaffTitle" type = "xsd:string"/>
  <xsd:element name = "PUBLICATION" type = "xsd:PUBLICATION_Type" minoccurs="0"
    maxOccurs= "unbounded"/>
  <!—without maxoccurs, if it is only a single "part"-->
  <xsd:attribute name = "StaffID" type = "xsd:ID" use = "required"/>
</xsd:complexType>

Transformation into ORDB (Step 2):

CREATE TYPE PublicationType
   (Title CHARACTER VARYING(200),
    Year INTEGER,
    Content CHARACTER VARYING(4000))
/
CREATE TABLE Staff
   (StaffID CHARACTER VARYING(5) CONSTRAINT StaffID_pk PRIMARY KEY,
    StaffName CHARACTER VARYING(40),
    StaffTitle CHARACTER VARYING(25),
    Publication MULTISET (PublicationType)
   -- Dependent DependentType for single aggregation);
```

have an element of the ‘part’ complex type. Having done this, the ‘part’ type can actually be used inside another ‘whole’ complex type (shareable). To preserve the collection, we use the XML Schema syntax **maxOccurs = ‘unbounded’**.

```
<xsd:complexType name = “PART_Type”>
        ...
</xsd:complexType>

<xsd:complexType name = “WHOLE_Type”>
       <xsd:element name = “PART_Name” type = “xsd:PART_Type” maxOccurs=
    “unbounded”/>...
</xsd:complexType>
```

In the second transformation, we map the XML Schema to the ORDB in the form of UDT collection attribute. The mapping is straightforward. The ‘part’ complex type is mapped as a UDT and the ‘whole’ complex type is mapped as a table with one attribute formed by the ‘part’ type. We use the SQL syntax **TABLE (…Attribute MULTISET<ARRAY[ ]> (UDT_TYPE))**. If for example, the ‘whole’ can only have one ‘part’ component, we do not use the collection. We use the SQL Syntax **TABLE (…Attribute UDT_TYPE)**.

4.2. Proposed method for composition relationship

Frequently, a composition is categorized as an aggregation. It is a ‘part-of’ relationship where the ‘part’ components are non-shareable and their existence depends on the ‘whole’ component. Therefore, we need to exclusively define the ‘part’ component inside the ‘whole’ component.

In the first step (see Fig. 5), we map the ‘part’ component as the complex type inside the ‘whole’ component. This process prevented other components from owning this particular ‘part’ component. For the second step, we directly map the outer complex type as the table or a type and the inner complex type as the row type attribute. For multiple-row, we use multiset data types standardized in SQL4 [14]. Another type of collection is also possible depending on the requirements for ordering semantic. The mapping rules are shown below [20].

4.2.1. Rule 2

Step 1: For two types namely, T_1 and T_2 with elements/attributes (A,B) and (M,N), respectively, if T_1 can be composed by non-shareable and existence-dependent T_2, implement T_1 as a complex type and T_2 as an inner complex type with maxoccurs constraint set to unbound for cardinality more than one.

Step 2: For two complex types namely, T_1 (A,B) and T_2 (M,N), if T_1 can be composed by shareable and existence-dependent T_2, implement T_2 as a single ROW or collection of ROW attribute of table or type T_1. Transformation result is Type|Table $T_1(A, B, \mathrm{Row}_0^n\ T_2(M, N))$.

4.2.2. Example 2

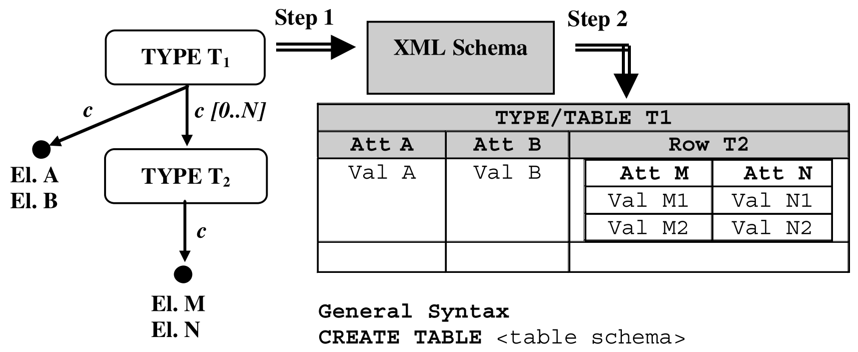
Continuing Example 1, now we want to extend the relationship between *publication* and *content*. Type *publication* is the composition of type *content* (see Fig. 4). The latter type can only exist inside type *publication*. The composition type will be mapped into single or collection of ROW in ORDB table or type.

```
Transformation into XML Schema (Step 1):

<xsd:complexType name = “PUBLICATION_Type”>
  <xsd:sequence>
    <xsd:element name = “Title” type = “xsd:string”/>
    <xsd:element name = “Year” type = “xsd:integer”/ >
    <xsd:element name = “CONTENT” minOccurs = “0” maxOccurs= “unbounded”/>
    <!—without maxoccurs, if it is only a single “part”-->
     <xsd:complexType>
      <xsd:sequence>
        <xsd:element name = “SectionNo” type = “xsd:integer” minOccurs = “0”
      maxoccur=”unbounded”/>
        <xsd:element name = “SectionContent” type = “xsd:string” minOccurs = “0”
      maxoccur= “unbounded”/>
      </xsd:sequence>
     </xsd:complexType>
    </element>
  </xsd:sequence>
</xsd:complexType>

Transformation into ORDB (Step 2):

CREATE TYPE PublicationType
   (Title CHARACTER VARYING(200),
    Year INTEGER,
    Content MULTISET (ROW (SectionNo INTEGER, SectionContent CHARACTER VARYING (4000)))
   -- Content ROW (SectionNo NUMBER, SectionContent CHARACTER VARYING (4000)) for single
      composition);
```



```
General Syntax
CREATE TABLE <table schema>
(att_i     data type, ....,
 att_i+1   <MULTISET|ARRAY> ROW
                     (att_(i+1)j data type, ....));
```

Fig. 5. Composition transformation methodology.

In the first transformation, we come up with the XML Schema where the 'part' component is defined as a complex type inside the 'whole' type element. By doing this, we avoid other element type to share the particular 'part' component (non-shareable constraint). We also ensure that on removal of the 'whole' type, we remove all 'part' components that are defined inside it (existence-dependent constraint). To preserve the collection, we use the XML Schema syntax **maxOccurs = "unbounded"**.

```
<xsd:complexType name = "WHOLE_Type">...
  <xsd:element name = "PART_Name" maxOccurs="unbounded"/>>
        <xsd:complexType>...
        </xsd:complexType>
  </xsd:complexType>
```

In the second step, we map the outer complex type as the type in the ORDB and the inner complex type as the ROW attribute. To preserve the collection, we implement the ROW as a collection with this syntax **TABLE (...MULTISET < ARRAY[ ] > (ROW( ))**. If, for example, the 'whole' can only have one part component, we do not use the collection. We use the SQL Syntax **TABLE (...Attribute ROW( ))**.

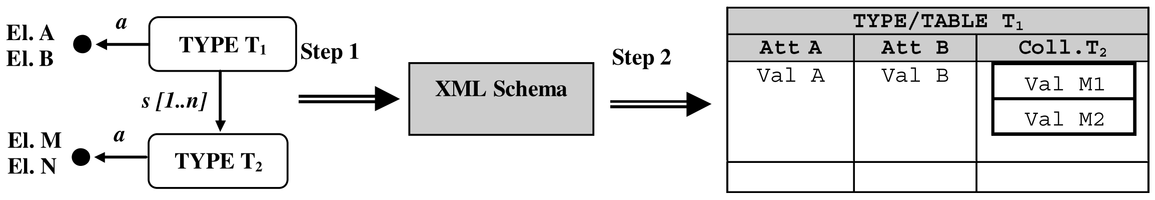
4.3. Proposed methods for association relationship

In this section, we propose the mapping methods of the association relationship in the XML data into the ORDB using the new complex structures [19]. We will only cover the association with 'many' cardinality: 1:*N* and *N*:*N*. We have not included 1:1 cardinality because the mapping will be very similar to the mapping of 1:*N* association, without involving collection data type.

As with the previous sections, the proposed method is divided into two steps. In the first step, we map the conceptual model in the semantic network into the XML Schema. The associating types will become the complex types. In the second step we map the complex types as the tables or types and the 'referential' object as the single or collections of type attribute.

4.3.1. One:many association

For the 1:*N* association relationship, the reference can be stored as a collection inside the type that has 'one' side. In usual practice, the XML Schema mapping to ORDB is not straightforward because the location of the



```
General Syntax

CREATE TABLE <T2 table schema>
(att_i     data_type, ....,
 att_i+m   data_type);

CREATE TABLE <T1 table schema>
(att_j     data_type, ....,
 att_j+n   MULTISET|ARRAY (T2 Key data_type));
```

TYPE/TABLE T_2	
Att M	Att N
Val M_1	Val N_1
Val M_2	Val N_2

Fig. 6. 1:*N* association transformation.

reference key in both schemas is different. Our method proposes a more straightforward mapping, since the reference type is always located in the type that has 'one' side (see Fig. 6).

4.3.1.2. Example 3. From the running example (see Fig. 1), we use the example of association between *staff* and *office* F (see Fig. 7). The reference element/attribute from 'one' to 'many' side type will be mapped as collections attribute in ORDB table.

```
Transformation into XML Schema (Step 1):

<xsd:complexType name = "STAFF_Type">
  <xsd:element name = "StaffName" type = "xsd:string"/>
  <xsd:element name = "StaffTitle" type = "xsd:string"/>
  <xsd:attribute name = "StaffID" type = "xsd:ID" use = "required"/>
</xsd:complexType>

<xsd:complexType name = "OFFICE_Type">
  <xsd:element name = "OfficeLoc" type = "xsd:string"/>
  <xsd:element name = "OfficePhone" type = "xsd:string"/>
  <xsd:element name = "StaffID" type = "xsd:IDREF" maxoccurs="unbounded"/>
  <xsd:attribute name = "OfficeNumber" type = "xsd:ID" use = "required"/>
</xsd:complexType>

<key name='StaffID_Key'>
    <selector xpath = "FACULTY/STAFF">
    <field xpath="@StaffID"/></key>

<key name='OfficeNumber_Key'>
    <selector xpath = "FACULTY/OFFICE">
    <field xpath="@OfficeNumber"/></key>

<keyref name='StaffID_Office_Ref" refer="StaffID_Key">
    <selector xpath = "FACULTY/OFFICE">
    <field xpath="StaffID"/></keyref>

Transformation into ORDB (Step 2):

CREATE TABLE Staff
   (StaffID CHARACTER VARYING(5) CONSTRAINT StaffID_pk PRIMARY KEY,
    StaffName CHARACTER VARYING(40),
    StaffTitle CHARACTER VARYING(25));

CREATE TABLE Office
   (OfficeNumber CHARACTER VARYING(5) CONSTRAINT OfficeNumber_pk PRIMARY KEY,
    OfficeLoc CHARACTER VARYING(20),
    OfficePhone CHARACTER VARYING(10),
    StaffID MULTISET(CHARACTER VARYING(5)));
```

4.3.1.1. Rule 3.

Step 1: For two types namely T_1 (A,B) and T_2 (M,N), if T_1 and T_2 has 1:N association relationship, implement both as complex types with T_1 has a collection of reference to T_2.

Step 2: For two complex types namely T_1 (A,B) and T_2 (M,N), if T_1 holds collection of reference to T_2, implement both as type/tables with T_1 has a collection attribute refer to T_2. Transformation result is Type|Table $T_1(A, B,\ T_2\_\mathrm{Key}_1^n)$ and Type|Table T_2 (M,N).

```
<xsd:complexType name = "ONE_Type">...
    </xsd:complexType>

<xsd:complexType name = "MANY_Type">
  <xsd:attribute name = "ONE_Key" ... maxOccurs= "unbounded"/>...
</xsd:complexType>

<key name="ONE_Key">
    <selector xpath = "ONE_Type">...</key>

<keyref name="ONE_Key_Ref" refer="ONE_Key">
    <selector xpath = "MANY_Type">...</keyref>
```

In our method, we utilize the collection to store the relationship between two types. For the first transformation we come up with two complex types. In the 'one' complex type we will have collection of element of the 'many' complex type. To preserve the collection, we use XML Schema syntax **maxOccurs = "unbounded"**. And to maintain the relationship, we use **key** and **keyref** in addition to **ID** and **IDREF**. Using the formers, we enable one to specify scope within which uniqueness applies [28].

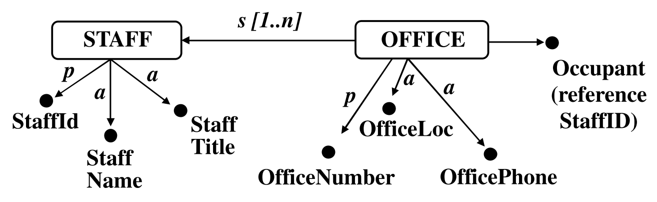


Fig. 7. 1:*N* association example.

In the second step, we map the complex types as the tables in the ORDB. In the 'many' type, we have collection attribute consisting of attribute key from the 'one' type. If we have a single key we will use a collection of simple data type. Otherwise, we will have a collection of ROW type. To preserve the collection the implementation syntax is **TABLE (**...**MULTISET**<**ARRAY[ ]**>**(SIMPLE_TYPE|ROW( ))**.

We have one shortcoming in using collection in the second step of association relationship. At present, we cannot use current SQL to embed integrity constraint checking in ORDB. As we know, in traditional methods we can include the foreign key or REF and then define the integrity constraint checking such as ON DELETE CASCADE, ON UPDATE NULLIFY, etc. We still cannot apply this for collection attribute. Nevertheless, it does not mean we cannot have integrity constraint checking for our methods. Triggers and embedded routines are available in ORDB to enforce this task.

4.3.2. Many:many association

In the *N*:*N* association, the reference can be stored as a collection inside one of the associated type (see Fig. 8). Our method proposes a different way of implementing *N*:*N* association because we do not require storing the relationship in a separate table.

4.3.2.1. Rule 4.

Step 1: For two types T_1 (A,B) and T_2 (M,N) have N:N association relationship in T_3 (X), implement both as complex types with T_1 has a collection of T_3 and reference to T_2.

Step 2: For two complex types namely T_1 (A,B) and T_2 (M,N) have N:N relationship, if T_1 has the collection of the relationship and the reference to T_2 implement

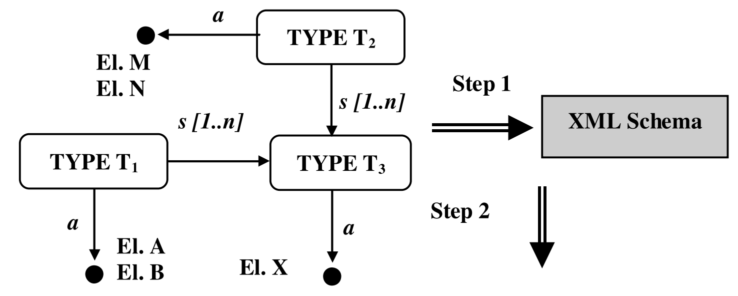


TABLE T_1			
Att A	Att B	Coll. T_2 Key and T_3	
		T2 Key	T3 Att X
Val A	Val B	Val M_1	Val X_1
		Val M_2	Val X_2

TABLE T_2	
Att M	Att N
Val M_1	Val N_1
Val M_2	Val N_2

```
General Syntax
CREATE TABLE <T2 table schema>
(att_i    data_type, ....,
 att_i+m  data_type);
```

```
CREATE TABLE <T1 table schema>
(att_j    data_type, ....,
 att_j+n  MULTISET|ARRAY ROW
          (att_i    data type, ....,
           att_h    data type, ....,
           att_h+1  data type);
```

Fig. 8. *N*:*N* association transformation.

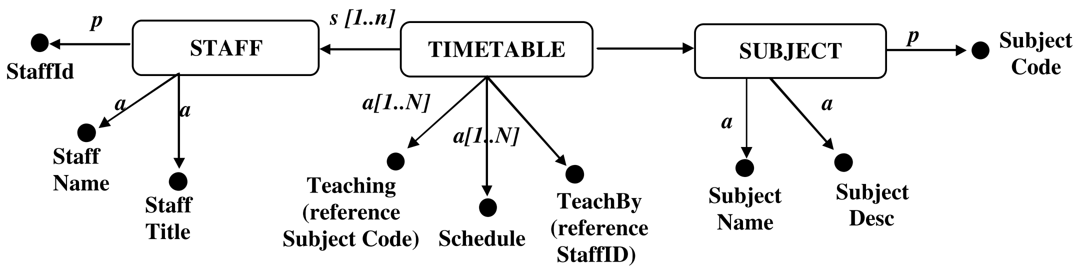


Fig. 9. STUDENT–SUBJECT association example.

both as tables with T_1 has collection ROW attribute. Transformation results are Type|Table $T_1(A, B, \mathrm{Row}_1^n\ (T_2\ \mathrm{Key}, T_3(X)))$ and Type/Table T_2 (M,N).

4.3.2.2. Example 4. From the running example (see Fig. 1), we use the example of association between *staff* and *subject* (see Fig. 9). In this *N*:*N* relationship, the reference of one type together with the relationships attributes will be mapped as collection in another type table.

```
Transformation into XML Schema (Step 1):

<xsd:complexType name = "STAFF_Type">
  <xsd:element name = "StaffName" type = "xsd:string"/>
  <xsd:element name = "StaffTitle" type = "xsd:string"/>
  <xsd:attribute name = "StaffID" type = "xsd:ID" use = "required"/>
</xsd:complexType>

<xsd:complexType name = "SUBJECT_Type">
  <xsd:element name = "SubjectName" type = "xsd:string"/>
  <xsd:element name = "SubjectDesc" type = "xsd:string"/>
  <xsd:element name = "TIMETABLE_Type" maxOccurs= "unbounded"/>
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name = "Schedule" type = "xsd:string"/>
     <xsd:attribute name = "StaffID" type = "xsd:IDREF"/>
    </xsd:sequence>
   </xsd:complexType>
  </element>
  <xsd:attribute name = "SubjectCode" type = "xsd:ID" use = "required"/>
</xsd:complexType>

<key name='SubjectCode_Key'>
    <selector xpath = "FACULTY/SUBJECT">
    <field xpath="@SubjectCode"/></key>

<keyref name='StaffID_Timetable_Ref" refer="StaffID_Key">
    <selector xpath = "FACULTY/SUBJECT/TIMETABLE">
    <field xpath="StaffID"/></keyref>

Transformation into ORDB (Step 2):

CREATE TABLE Staff
   (StaffID CHARACTER VARYING(5) CONSTRAINT StaffID_pk PRIMARY KEY,
    StaffName CHARACTER VARYING(40),
    StaffTitle CHARACTER VARYING(25));

CREATE TABLE Subject
   (SubjectCode CHARACTER VARYING(5) CONSTRAINT SubjectCode_pk PRIMARY KEY,
    SubjectName CHARACTER VARYING(50),
    SubjectDesc CHARACTER VARYING (400),
    Teachby MULTISET (ROW
        (StaffID CHARACTER VARYING(5),
         Schedule CHARACTER VARYING(50))));
```

In the first step, we map the associated types into two separate complex types. In one of the complex types, we include the key to the other complex type as well as the relationship elements/attributes. Same as 1:*N* association, we use **maxOccurs = "unbounded"** with **key** and **keyref** to preserve the collection and the referential constraint. The difference is now we have additional elements/attributes that come up with the relationship.

In the second step, we will be likely to have a collection of ROW attribute inside one of the 'many' side table. It is because we need to include the key to the other 'many' side table and the additional relationship attributes. To preserve the collection, we implement the collection with this syntax **TABLE (…MULTISET <ARRAY[ ]> (ROW( ))**.

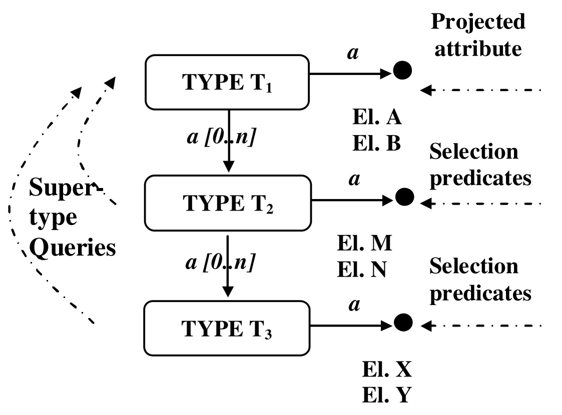
5. Proposed query techniques

In this section, we propose different SQL queries for XML data stored in ORDB. They will follow our proposed transformation methods that have been described in the previous sections. We differentiate the query based on the target of projection and the selection predicate. Along with aggregation and composition relationships, we describe sub-type query and super-type query. Also with association relationship, we describe referencing query and dereferencing query.

5.1. Super-type and sub-type query

Super-type query is a query which retrieves the information of the super-type table with the selection predicates originated at sub-type complex structures attributes. Fig. 10 shows the flow of query in a sub-class query along with the general syntax.

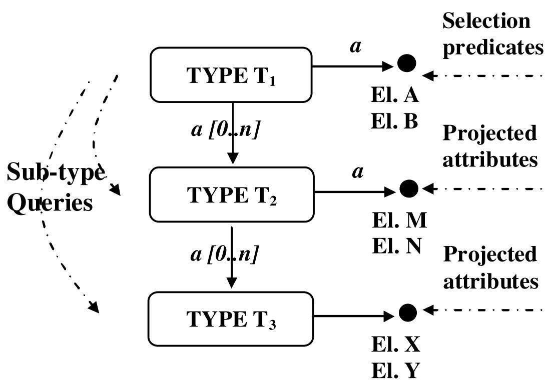
Sub-type query is a query which retrieves the information of the complex structure with the selection predicates originated at the super-type class attributes. Fig. 11 shows the flow of



```
SELECT <a_1.super-type attributes>
FROM <super-type table a_1>,
       TABLE <a_1.sub-type name a_2>,...
       TABLE <a_{n-1}.sub-type name a_n>
WHERE <[a_2..]|[a_n..]sub-type attributes
       = &input_selection_predicates>

where: 1 <= n
        a_i is super-type table
        a_2 is sub-type of table a_1
        a_n is sub-type of sub-type a _{n-1}
```

Fig. 10. Super-Type query flow.



```
SELECT <[a_2..]|[a_n..]sub-type
         attributes>
FROM <super-type table a_1>,
       TABLE <a_1.sub-type name a_2>,...
       TABLE <a_{n-1}.sub-type name a_n>
WHERE <a_1.super-type attribute =
       &input_selection_predicates>

      where: 1 <= n
        a_i is super-type table
        a_2 is sub-type of table a_1
        a_n is sub-type of sub-type a_{n-1}
```

Fig. 11. Sub-type query flow.

query in a sub-class query and the general syntax. Note that SQL provides double dot notation [9] to specify a content of a complex structures such as ROW type or UDT in a table.

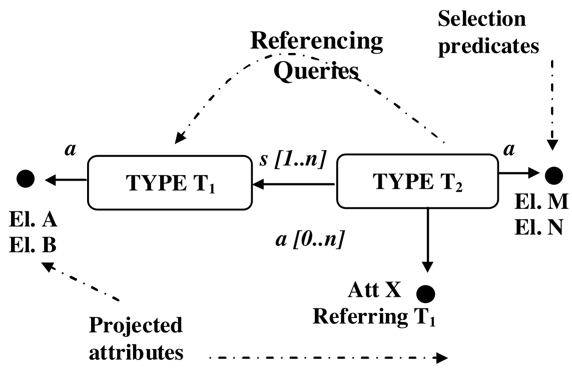
5.2. Referencing query and dereferencing query

Referencing query is a query which retrieves the information of the referred type (implemented in the complex structures) with the selection predicates originated at the type that holds the reference. Fig. 12 shows the flow of query in a referencing query and the general syntax. The general syntax is applicable if the referencing attribute is in the direct complex attribute of the referencing table. Of course, there is a possibility of the referencing attributes appearing below this level.

Dereferencing query is a query to retrieve the information of the referencing type with the selection predicates originated at the type that is being referred. Fig. 13 shows the flow of query in a referencing query and the general syntax.

6. Evaluation

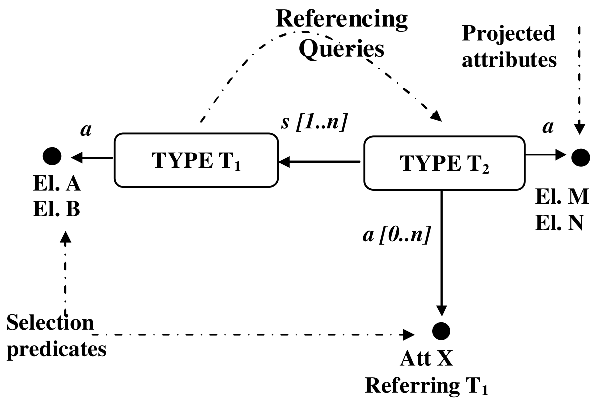
In this section, we will evaluate our proposed methodologies of storing XML Documents in ORDB using complex structures. We divide the section into quantitative and qualitative evaluation. The first evaluation aims to show how



```
SELECT <b.referred table attributes>
FROM <referencing table a>,
       <referred table b>,
       TABLE <a.reference c>
WHERE <c..referencing attributes
       = b.referred attribute>
AND <a.referencing table attributes
     = &input_selection_predicates>

where: a is the referencing table
       b is the referred table
       c is the referencing
         attribute collection
```

Fig. 12. Referencing query flow.



```
SELECT <a.referencing table
attributes>
FROM <referencing table a>,
       <referred table b>
       TABLE <a.reference c>
WHERE <c..referencing attributes
       = b.referred attribute>
AND <b.referred table attributes
     = &input_selection_predicates>

where: a is the referencing table
       b is the referred table
       c is the referencing
         attribute collection
```

Fig. 13. Dereferencing query flow.

our proposed methodologies reduce the query cost compared to the current practice. The second evaluation aims to show the contribution of our works in a qualitative manner by highlighting the strength of our methods compared to the current implementation.

6.1. Quantitative analysis

The straightforward way to perform quantitative analysis is by running the queries in two different scenarios, the current practice and the proposed method. If we decide to do that, we have to use one of the available ORDBMS for implementation. However, it is inevitable that different product implement SQL features differently and very frequently not all SQL recommendations are followed through by these ORDBMS.

From the formula, it is known that the evaluation will consider the number of page accesses only. Also, the equation suggests that two important parameters are record size and number of records.

6.1.1. Super-type and sub-type query evaluation

For this query we use the example of aggregation between *Staff* and the *Publication*. Without using ORDB complex structures, we implement these two types into three tables. The last table holds the keys to the other two tables. Our proposed method implements this composition using a collection of UDT. We will use a simple super-type query, to retrieve the staff detail for the publication with title 'XML Updates' and is published in the year 2005.

Without ORDB Complex Structures:	*Proposed Transformation:*
STAFF (StaffID, StaffName, StaffTitle)	*Staff (StaffID, Year, MULTISET (*
PUBLICATION (Title, Year, Content)	*PUBLICATIONTYPE (Title, Year, Content)))*
STAFFPUBL (StaffID, Title)	
SELECT S.StaffTitle, S.StaffName	*SELECT S.StaffTitle, S.StaffName*
FROM Publication P, Staff S, StaffPubl SP	*FROM Staff S, TABLE(S.Publication) P*
WHERE S.StaffID = SP.StaffID	*WHERE P..Title = 'XML Updates'*
AND P.Title = SP.Title	*AND P..Year = 2005;*
AND P.Title = 'XML Updates'	
AND P.Year = 2005;;	

To avoid a vendor-specific evaluation, we determined to use another approach of quantitative evaluation, which is by using a cost model. The cost model will be based on the number of I/O accesses as it is considered much more expensive than the CPU processing cost (e.g. storage cost, computation cost, etc.) [22].

In this cost model, we calculate the retrieval cost of the query. Each object-relational table T has a set of records $r=(r_1, r_2, \ldots, r_n)$ which are stored in a set of pages $p=(p_1, p_2, \ldots, p_3)$. If we assume that the tables involved in a query operation are scanned separately, we can formulate the cost as follows:

$$PageCost = Round\left\{\frac{R}{Trunc(pagesize/S)}\right\} = \left\{\frac{RS}{pagesize}\right\},$$

where R is the number of records and S is the record size.

Quantitatively, the record size and number of records for flat table implementation are as follows

$$S_{\mathrm{T}} = \left(S_{StaffID} + \sum_{g=0}^{k} S_{StaffAtts}\right) + \left(S_{Title} + \sum_{h=0}^{l} S_{PublAtts}\right) + (S_{StaffID} + S_{Title});$$

$$R_{\mathrm{T}} = \sum_{i=1}^{m} R_{StaffTable} + \sum_{j=1}^{n} R_{PublTable} + \sum_{j=1}^{n} R_{StaffPublTable},$$

where g and h are the number of attributes in the tables excluding the ID attribute, i and j are the number of rows in the tables and $\{k, l, m, n\} \geq 1$.

The record size and number of records for the proposed design are as follows:

$$S_{\mathrm{P}} = \left(S_{StaffID} + \sum_{g=0}^{k} S_{StaffAtts} + S_{Title} + \sum_{h=0}^{l} S_{PublAtts} \right);$$

$$R_{\mathrm{P}} = \sum_{i=1}^{m} R_{StaffTable} + \sum_{j=1}^{n} R_{PublTable},$$

where g and h are the number of attributes in the tables excluding the ID attribute, i and j are the number of rows in the tables and $\{k, l, m, n\} \geq 1$.

Now we can use the cost model for both mapping results

$$C_{\mathrm{T}} = \frac{\sum_{i=1}^{m} R_{StaffTable} + \sum_{j=1}^{n} R_{PublTable} + \sum_{j=1}^{n} R_{StaffPublTable}}{pagesize} \times \left(2S_{StaffID} + \sum_{g=0}^{k} S_{StaffAtts} \right) + \left(2S_{Title} + \sum_{h=0}^{l} S_{PublAtts} \right),$$

$$C_{\mathrm{P}} = \frac{\sum_{i=1}^{m} R_{StaffTable} + \sum_{j=1}^{n} R_{PublTable}}{pagesize} \times \left(S_{StaffID} + \sum_{g=0}^{k} S_{StaffAtts} + S_{Title} + \sum_{h=0}^{l} S_{PublAtts} \right)$$

The number of records component for the proposed method is smaller than the number of records component of the traditional method. Also, the record size component for the propose method is smaller than the record size component of the traditional method. From these two components, we can conclude that the cost of our proposed method is smaller than the cost of using an existing method.

To elaborate, in our database example, the record sizes for *Staff* and *Publication* attributes are 1 KB each. The number of records for *Staff* and *Publication* are 100 and 300. Assuming every publication is written by two staff, the number of records in *StaffPubl* will be 600. Assume that the record size for id is 0.01 KB each, each table page is 2 KB, and a record may not span more than a single page. The costs are $C_{\mathrm{T}} = (300 + 400 + 600) \times (2 \times 0.01 + 1 + 2 \times 0.01 + 1)/2 = 1326$ I/O accesses, and $C_{\mathrm{P}} = (100 + 300) \times (0.01 + 1 + 0.01 + 1)/2 = 404$ I/O accesses. The cost of the proposed method in this case is a third of the cost of using flat tables.

We also found a smaller cost for sub-type query using our proposed method. Due to the page limitation, we do not show this result in this paper.

6.1.2. Referencing and dereferencing query evaluation

For this query, we use the example of 1:*N* association relationship between *Staff* and the *Office*. With or without complex structures, we will use two separate tables. The interesting fact is that a referencing query in our proposed implementation is actually a dereferencing query in traditional implementation. This is because the location of the reference keys is different.

We will use a referencing query, to retrieve the staff ID of the office location 'Bundoora E1'.

Without ORDB Complex Structures:	Proposed Transformation:
OFFICE (OfficeNumber, OfficeLoc,	OFFICE (OfficeNumber, OfficeLoc,
OfficePhone)	OfficePhone, MULTISET (StaffID))
STAFF (StaffID, StaffName, StaffTitle,	
OfficeNumber)	
SELECT S.StaffID	SELECT S..StaffID
FROM Staff S, Office O	FROM Office O, TABLE(O.Staff) S
WHERE S.OfficeNumber = O.OfficeNumber	WHERE O.OfficeLoc = 'Bundoora E1';
AND O.OfficeLoc = 'Bundoora E1';	

Quantitatively, the record size and number of records for flat table implementation are as follows

$$S_{\mathrm{T}} = \left(S_{OfficeNumber} + \sum_{g=0}^{k} S_{OfficeAtts} \right) + \left(S_{StaffID} + \sum_{h=0}^{l} S_{StaffAtts} + S_{OfficeNumber} \right);$$

$$R_{\mathrm{T}} = \sum_{i=1}^{m} R_{OfficeTable} + \sum_{j=1}^{n} R_{StaffTable},$$

where g and h are the number of attributes in the tables excluding the ID attribute, i and j are the number of rows in the tables and $\{k, l, m, n\} \geq 1$.

The record size and number of records for proposed design are as follows

$$S_{\mathrm{P}} = \left(S_{OfficeNumber} + \sum_{g=0}^{k} S_{OfficeAtts} + S_{StaffID} \right);$$

$$R_{\mathrm{T}} = \sum_{i=1}^{m} R_{OfficeTable} + \sum_{j=1}^{n} R_{StaffIDinOfficefTable},$$

where g is the number of attributes in the table excluding the ID attribute, i and j are the number of rows in the table and in the collection of referencing attribute and $\{k, l, m, n\} \geq 1$.

Now we can use the cost model for both mapping results.

$$C_{\mathrm{T}} = \frac{\sum_{i=1}^{m} R_{OfficeTable} + \sum_{j=1}^{n} R_{StaffTable}}{pagesize} \times \left(2S_{OfficeNumber} + S_{StaffID} + \sum_{g=0}^{k} S_{OfficeAtts} + \sum_{h=0}^{l} S_{StaffAtts} \right),$$

$$C_{\mathrm{P}} = \frac{\sum_{i=1}^{m} R_{OfficeTable} + \sum_{j=1}^{n} R_{StaffIDinOfficeTable}}{pagesize} \times \left(S_{OfficeNumber} + \sum_{g=0}^{k} S_{OfficeAtts} + S_{StaffID} \right).$$

Even though the number of record component for the proposed method does not differ significantly to the number of record component in traditional method, the record size component for our proposed method is significantly smaller than the record size component of the traditional method. From the difference in the record size component, we can conclude that the cost of our proposed method is smaller than the cost of using an existing method.

To elaborate, in our database example the record sizes for *Staff* and *Office* attributes are 1 KB each. The number of records for *Staff* and *Office* are 100 and 50. Assuming every office is occupied by two staff, the number of *StaffId* in Office table will be 100 (for our proposed implementation). Assume that record size for id is 0.01 KB each, each table page is 2 KB, and a record may not span more than a single page. The costs are $C_{\mathrm{T}}=(100+50)\times(3\times0.01+1+1)/2=152.25$ I/O accesses, and $C_{\mathrm{P}}=(100+100)\times(2\times0.01+1)/2=102$ I/O accesses. The cost of the proposed method in this case is two-thirds of the cost of using flat tables.

We also found a smaller cost for dereferencing query using our proposed method. Due to the page limitation, we do not show this result in this paper.

6.2. Qualitative analysis

From the previous evaluation, we have found that the implementation using our proposed method reduces the costs of many queries. In this section, we use a qualitative analysis to highlight the strengths of our methods.

6.2.1. Support for set-valued attribute

Shanmugasundaram et al. [26] points out that one of the main weaknesses of using flat tables to store XML documents is the inability to support set-valued attributes. Set-valued attributes are useful in storing sub-elements without fragmentation and in generating complex XML results. With the existence of complex structures in ORDB, our methodologies are able to accommodate set-valued attributes in one of the collection types. This practice enables us to map more XML tree structures and reduces, if not eliminates the fragmentation of the data in many relations.

Currently, SQL only allows array and multiset. We may use arrays for set-valued attributes with an addition of an index. In further development, SQL will enable set collection [9].

6.2.2. Captures relationship semantic

Our mapping methodologies capture the real meaning of relationships in XML Document. Without complex structures, the XML document is stored in flat tables and all of the relationships are treated as association relationships. For example, with flat table implementation a composition is treated as a 1:*N* association. With our methods, we store the sub-types inside the super-type and thus the existence dependent meaning remains intact.

6.2.3. Support for various query types

Our mapping methodologies enable the use of different queries. In flat table implementation, we cannot have super-type and sub-type query because the super-type and sub-type inside the XML document are flattened. Without knowledge of the conceptual design, we would not know which table stores the super-type and which tables store the sup-types. With our methodologies, these queries can be implemented with ease because the conceptual structures are well kept in the physical implementation

6.2.4. Multiple query optimisations

Query in a flat table only applies to a simple path direction in an XML document. If we want to generate an XML document that requires complex paths, we need multiple simple paths to be converted into simple SQL queries. This, of course, deals with more costly operation such as scan and joins. With our implementation, we can support more complex queries due to our more complex structures. A complex path query can be converted more easily into our SQL query and thus, requires less scans and joins.

7. Conclusion

Compared to relational data, XML documents have considerably more complex structures. Storing the XML data in a relational database, existing works implement the complex structure as flat tables. This practice has diminished the conceptual model semantic. With the existence of some complex structures data types in ORDB, we propose a novel implementation of XML repository in ORDB. In this paper, we demonstrate how different relationship structures such as aggregation, composition and different association can be preserved in the implementation by using user-defined type, row type and collection type in ORDB.

Unlike some works in transformation, our proposed methods cover two straightforward mapping steps, spanned from the conceptual model to the implementation into tables. In logical level, we propose the mapping of Semantic Network Diagram into XML Schema. In implementation level, we map the XML Schema into ORDB schema tables with complex data types. By doing this, the results maintain the semantic stated in the conceptual level. In addition, using the complex data types, we have utilized the rich facility in ORDB.

Further, we also propose the query to retrieve the XML data. These queries are further differentiated based on the projection target and the selection predicates into four: super-type query, sub-type query, referencing query and dereferencing query.

Our evaluations show that our proposed methods lead to the development of a better alternative for XML data repository. Not only does it generate less query cost, it can also be used for more various queries and it maintains the real conceptual semantics further down to the implementation.

Acknowledgements

We would like to thank the anonymous reviewers for their thorough and comprehensive reviews. Their valuable suggestions have helped us in revising this paper.

References

[1] A. Almarimi, J. Pokorny, A mediation layer for heterogeneous XML schemas, International Journal of Web Information Systems 1 (1) (2005) 25–32.

[2] R. Bourret, Mapping DTDs to Databases, 2004, Available in: http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html

[3] R.C.G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, F. Velez (Eds.), The Object Database Standard: ODMG 3.0, Morgan Kaufmann, San Francisco, CA, 2000.

[4] CODASYL Database Task Group, Data Base Task Group Report, 1971.

[5] R. Elmasri, S.B. Navathe, Fundamentals of Database Systems, Addison-Wesley, Boston, MA, 2002.

[6] L. Feng, E. Chang, T. Dillon, A semantic network-based design methodology for XML documents, ACM TOIS 20 (4) (2002) 390–421.

[7] L. Feng, T.S. Dillon, An XML-enabled data mining query language: XML-DMQL, International Journal of Business Intelligence and Data Mining 1 (1) (2005) 22–41.

[8] D. Florescu, D. Kossmann, Storing and querying XML data using an RDMBS, IEEE Data Engineering Bulletin 22 (3) (1999) 27–34.

[9] P. Fortier, SQL3 Implementing the SQL Foundation Standard, McGraw-Hill, New York, 1999.

[10] W.-S. Han, K.-H. Lee, B.S. Lee, An XML storage system for object-oriented/object-relational DBMSs, Journal of Object Technology 2 (1) (2003) 113–126.

[11] Ipedo, Ipedo XML Database, Available in: http://www.ipedo.com/html/products.html, 2004.

[12] G. Jaeschke, H.-J. Schek, Remarks on the algebra of non first normal form relations, Proceedings of the ACM PODS 1982, ACM Press, Los Angeles, CA, 1982, pp. 124–138.

[13] 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 Journal 11 (4) (2002) 279–291.

[14] M. Klettke, H. Meyer, XML and object-relational database systems—enhancing structural mappings based on statistics, Proceedings of the WebDB 2000, Springer, Dallas, TX, 2000, pp. 151–170.

[15] 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, Boston, MA, 2003, pp. 43–68.

[16] J. Melton (Ed.), Database Language SQL—Part 2 Foundation, ISO-ANSI WD 9072-2, International Organization for Standardization, Working Group WG3, August 2002.

[17] V. Nassis, R. Rajagopalapillai, T.S. Dillon, J.W. Rahayu, Conceptual and systematic design approach for XML document warehouses, International Journal of Data Warehousing and Mining 1 (3) (2005) 63–87.

[18] E. Pardede, J.W. Rahayu, D. Taniar, New SQL standard for object-relational database applications, Proceedings of SIIT 2003, IEEE, Delft, The Netherlands, 2003, pp. 191–203.

[19] E. Pardede, J.W. Rahayu, D. Taniar, On using collection for aggregation and association relationships in XML object-relational storage, Proceedings of ACM SAC 2004, ACM Press, Nicosia, Cyprus, 2004, pp. 703–710.

[20] E. Pardede, J.W. Rahayu, D. Taniar, Preserving composition in XML object relational storage, Proceedings of AINA 2005, vol. 2, IEEE Computer Society, Taipei, Taiwan, 2005, pp. 695–700.

[21] E. Pardede, J.W. Rahayu, D. Taniar, Preserving conceptual constraints during XML updates, International Journal of Web Information Systems 1 (2) (2005) 65–82.

[22] J.W. Rahayu, E. Chang, T.S. Dillon, D. Taniar, Performance evaluation of the object-relational transformation methodology, Data and Knowledge Engineering 38 (2001) 265–300.

[23] M.A. Roth, H.F. Korth, The design of 1NF relational databases into nested normal form, Proceedings of SIGMOD 1987, ACM Press, San Francisco, CA, 1987, pp. 143–159.

[24] L.I. Rusu, J.W. Rahayu, D. Taniar, A methodology for building XML data warehouses, International Journal of Data Warehousing and Mining 1 (2) (2005) 23–48.

[25] K.-D. Schewe, B. Thalheim, The co-design approach to web information systems development, International Journal of Web Information Systems 1 (1) (2005) 5–14.

[26] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, J.F. Naughton, Relational databases for querying XML documents: limitations and opportunities, Proceedings of VLDB 1999, Morgan Kauffman, Edinburgh, Scotland, 1999, pp. 302–314.

[27] Software AG, TAMINO, Number One in XML Management, 2004, Available in: http://www1.softwareag.com/corporate/products/tamino/default.asp.

[28] E.V.-D. Vlist, XML Schema, O'Reilly, Sebastopol, 2002.

[29] N.D. Widjaya, D. Taniar, J.W. Rahayu, E. Pardede, Association relationship transformation of XML schemas to object-relational databases, Proceedings of iiWAS 2002, SCS Publishing House, Bandung, Indonesia, 2002, pp. 135–142.

[30] N.D. Widjaya, D. Taniar, J.W. Rahayu, Aggregation transformation of XML schemas to object-relational databases, Proceedings of IICS 2003, Springer, Leipzig, Germany, 2003, pp. 251–262.

[31] R. Xiaou, T.S. Dillon, E. Chang, E.L. Feng, Modeling and transformation of object-oriented conceptual models into XML schema, Proceedings of DEXA 2001, Springer, Munich, Germany, 2001, pp. 795–804.