

Creating XML documents from relational data sources

Csar M. Vittori
Carina F. Dorneles
Carlos A. Heuser

Universidade Federal do Rio Grande do Sul - UFRGS

Instituto de Informtica

Caixa Postal 15064 - CEP 91501-970

Porto Alegre - RS - Brasil

e-mail: {cvittori—dorneles—heuser}@inf.ufrgs.br

Abstract

In various application, it may be necessary to obtain XML representation from data maintained in relational databases. This paper presents a language called XML/SQL which is aimed at the specification of XML documents extracted from the results of relational queries. XML/SQL allows the creation of documents of arbitrary structure regardless of the structure of the relational data source. Further, XML/SQL is a declarative language, i.e., no procedural specification of the result is required.

1 Introduction

XML [BPSM98] is becoming a standard for the representation and exchange of data through the WEB. A large proportion of existing data are still stored in traditional databases. In many cases, it is necessary to transform data stored in traditional databases into the XML format. That happens, for example, when one wishes to generate WEB pages from the contents of a database, when it is necessary to exchange data stored in a database with other applications, or when applications process data internally using XML format.

This paper tackles the problem of transforming data stored in a relational databases into documents in XML format.

One approach to this problem is to use a XML query language, such as XML-QL [D⁺98], XQL [I⁺98], or Quilt [CRF00]. In that approach, the relational database model must be mapped into XML. The queries submitted in a XML query language must be mapped into SQL. One example of such approach is SilkRoute [FTS00]. This type of solution is suitable for users who are familiar with the XML model and its query language.

Another approach consists of using SQL queries and mapping the resulting relational views into the XML model. In that approach, the user knows the schema of the relational database. Such approach is suitable for users who are familiar with the relational model and the SQL language. This is the approach

followed here, as it is directed to programmers of applications that access relational databases.

Several tools that follow this approach have been proposed [Bou00]. In many of them, the structure of the resulting XML document is restricted by the structure of the relational database view. The structure of the resulting XML documents resembles the flat structure of relational views. No nesting of collections is allowed. Examples of tools that fit in this category are DB2XML [Vol99], Oracle XML SQL Utility [ORA99, Wai99], as well as extension of the APIs SAX and DOM for databases [Lad01].

Tools that allow the specification of resulting documents with nesting of collections are the ODBC2XML tool [ISR00] and the IBM DB2 XML Extender tool [CX00, IBM00].

The limitation of ODBC2XML is that the specification of the result is procedural and, for complex documents, involves the execution of interactive SQL queries. This can compromise the efficiency of the execution of the query, because the query optimizer of the relational DBMS is not used.

In the IBM DB2 XML Extender tool, each query consists of a single SELECT SQL command. This limits the expressive power of the language where dealing with several 1:n relationships involving the same entity (see example in Section 2.3).

In this paper, we describe XML/SQL, a language for creating XML documents from relational data sources. XML/SQL allows the specification of arbitrary XML documents with several levels of nesting, by means of a flexible mapping from relations to XML documents. The query itself as well as its result are XML documents.

This way, the XML/SQL language provides a flexible mechanism for querying and extracting data from relational sources. This allows the semi-automatic generation of relational wrappers [PGMW95] which produce XML views of a relational database.

This paper is structured as follows. Section 2 presents the XML/SQL language through examples given in order of increasing complexity. Initially (in Section 2.1), we give an example query without nesting which reflects the flat structure of the relational view. Afterwards, we present queries which involve nesting of collections (Section 2.2) and nesting of several collections in the same element (Section 2.3). The DTD of the XML/SQL language is given in Appendix A.

2 XML/SQL

We use the database shown in Figure 1 in the examples of XML/SQL queries that follow. That database comprises departments, professors, and courses. Each department is described by a department ID, a name, and an acronym. For each department, there are several courses associated with it, and each course is described by a course ID, a name, and the department ID to which it belongs. There exist several professors in each department, and professors are described by a professor ID, a name, and the department ID for which they work. Also, a course is taught by several professors, and one professor teaches several courses.

Professor		
IdProf	ProfName	IdDept
P1	Ann	D1
P2	John	D1
P3	Bill	D2

ProfCourse	
IdProf	IdCourse
P1	1
P1	2
P2	1

Department		
IdDept	DeptName	Acr
D1	Computer Science	CS
D2	Philosophy	

Course		
IdCourse	IdDept	CourseName
C1	D1	Database System
C2	D2	Philosophy I

DeptEvent	
IdDept	IdEvent
D1	E1
D1	E2
D2	E3

Event	
IdEvent	EventName
E1	Workshop
E2	Party
E3	Seminary
E4	Conference

Figure 1: Database used in examples

2.1 Creating documents with flat structure

An XML/SQL query is a valid XML document, according to the XML/SQL DTD presented in Appendix A (see structure in Figure 2).

```
<XMLSQL>
  <QUERY>
    ...
  </QUERY>
  <CONSTRUCT>
    ...
  </CONSTRUCT>
</XMLSQL>
```

Figure 2: Structure of an XML/SQL query

The root element is the XMLSQL element. This element contains two elements:

- the QUERY element, which encloses the SQL queries to the relational database, and
- the CONSTRUCT element, which contains the specification of the XML document to be created from the SQL queries.

The QUERY clause consists of a set of SQL instructions, which are executed over *base relations* generating *base views* as result. Each SQL instruction is enclosed within an SQL element.

The CONSTRUCT clause specifies the XML document resulting from the base views.

Using the database shown in Figure 1, the XML/SQL query below produces a list of departments.

```
<XMLSQL version="1.0">
  <QUERY>
    <SQL idsql="v1">SELECT d.iddept, d.deptname, d.acr
                      FROM department d
    </SQL>
  </QUERY>
```

```

</QUERY>
<CONSTRUCT>
  <LIST tagname="Departments" idsql="v1">
    <SEQUENCE tagname="Department">
      <ATOM tagname="Dept" source="iddept"/>
      <ATOM tagname="Name" source="deptname"/>
      <ATOM tagname="Acronym" source="acr" mandatory="yes"/>
    </SEQUENCE>
  </LIST>
</CONSTRUCT>
</XMLSQL>

```

In this query, the SQL instruction generates a base view whose schema is `v1(iddept, deptname, acr)`. This relation holds the ID, the name, and the acronym of each department. The `idsql` attribute of the SQL element identifies that base view.

The XML/SQL result produced by the previous query execution is as follows:

```

<Departments>
  <Department>
    <Dept>1</Dept>
    <Name>Computer Science</Name>
    <Acronym>CS</Acronym>
  </Department>
  <Department>
    <Dept>2</Dept>
    <Name>Philosophy</Name>
    <Acronym></Acronym>
  </Department>
</Departments>

```

The result is a list of departments. Each `Department` element corresponds to a department tuple in the base view. A `Department` element is composed of a sequence of elements which correspond to the base view attributes.

In the specification of the result of a XML/SQL query, the following *constructors* can be used:

- `LIST`: defines a list (collection) of elements of the same type;
- `SEQUENCE`: defines a sequence of elements, possibly of varying types;
- `ATOM`: defines a *text-only* element.

In the example query, the `tagname` attribute from the `LIST` constructor defines the name `Departments` for the element generated as root in the XML/SQL result. Similarly for the `SEQUENCE` and `ATOM` constructors, the `tagname` attribute gives names to elements. This attribute is required in all constructors. The `idsql` attribute from the `LIST` constructor makes reference to the base view which contains the corresponding data. In the particular case of that example, `idsql="v1"` makes reference to the base view which contains the department tuples. The `LIST` constructor generates a component element for each tuple in the base view.

In that same case, the `SEQUENCE` constructor specifies the element that will be generated for each tuple in the base view. In that example, each `SEQUENCE` corresponds to an element composed of three elements which are specified by the `ATOM` constructor.

The source attribute of the ATOM constructor makes reference to the attribute in the corresponding base view. In the example, the value for the `iddept` attribute in one line of the base view `v1` is used as content for a `Dept` element generated in the XML/SQL result.

The mandatory attribute in the ATOM constructor is used for handling null values. When the mandatory attribute is assigned `yes`, an element will always be generated in the result, regardless of the value of the attribute in the base view being `NULL` or not. When the mandatory attribute is assigned `no` (which is the “default” value), an element will only be generated in the query result in case the attribute value in the base view is different from `NULL`. For example, in the case of the `acr` attribute of the base view `v1`, the ATOM constructor’s mandatory attribute assigned value `yes` forces the generation of an Acronym element (see department 2 in the example).

A valid result specification is formed by a combination of the constructors defined above according to the XML/SQL DTD, given in the Appendix.

Following the XML formation rules, the result of a XML/SQL query execution has a single root element. As a consequence, the result specification always contains a single `LIST` constructor, which corresponds to the root of the XML/SQL result. This way, the XML/SQL result will always be a list.

A `LIST` constructor has a single child constructor which defines the type of the elements in the list. Three types of lists can be defined according to the child constructor:

- `LIST`: list of lists
- `SEQUENCE`: list of complex objects (a complex object is defined as a sequence of constructors)
- `ATOM`: list of values (textual elements)

A `SEQUENCE` constructor has an arbitrary sequence of child constructors. An `ATOM` constructor has no child constructors.

2.2 Nesting elements

In the previous example, the XML document reflects the planar structure of a relation. The root element is a list of sequences of atoms.

The XML/SQL language allows the creation of XML documents with arbitrary nesting of elements. As an example, consider the query below:

```
<QUERY>
  <SQL idsql="v2">SELECT d.iddept, d.deptname, c.idcourse, c.coursename
                    FROM department d, course c
                    WHERE d.iddept = c.iddept
                    ORDER BY d.iddept
  </SQL>
</QUERY>
```

This query retrieves, for each course, the department identifier, the department name, the course identifier and the course name. The result is sorted by department identifier.

Consider that we wish to write an XML/SQL query which, for the database in Figure 1, obtains the following result:

```

<Departments>
  <Department>
    <Dept>1</Dept>
    <Name>Computer Science</Name>
    <Courses>
      <Course>Database Systems</Course>
      <Course>Compilers</Course>
    </Courses>
  </Department>
  <Department>
    <Dept>2</Dept>
    <Name>Philosophy</Name>
    <Courses>
      <Course>Philosophy I</Course>
    </Courses>
  </Department>
</Departments>

```

In this document, for each department, the courses from that department are shown.

In order to get that result on the base view `v2` above, we need the `CONSTRUCTOR` clause below:

```

<CONSTRUCTOR>
<LIST tagname="Departments" idsql="v2" nestby="iddept">
  <SEQUENCE tagname="Department">
    <ATOM tagname="Dept" source="iddept"/>
    <ATOM tagname="Name" source="deptname"/>
    <LIST tagname="Courses" idsql="v2">
      <ATOM tagname="Course" source="coursename"/>
    </LIST>
  </SEQUENCE>
</LIST>
</CONSTRUCTOR>

```

In the example above, the `nestby` attribute of the `LIST Departments` constructor is the attribute's name (`iddept`) in the base view `v2` which identifies each `Department` object. For each value of this attribute in the base view, an element from the list will be generated; in this case, a `Department` element. That is, it is possible that an XML element will be generated from various tuples from the base view.

Note that in this case the use of the `ORDER BY` clause in the SQL query is required, so as to allow the identification of the tuples which compose an XML element, when it runs through the base view. The `nestby` attribute consists of a list of attribute names from the base view.

The generated `Department` element is a sequence comprising:

1. an atomic element `Dept` taken from the `iddept` attribute
2. an atomic element `Name` taken from the `deptname` attribute, as well as,
3. a list element named `Courses`.

This list is composed of atomic elements, one for each course from the respective department. As the specification of this list does not have the `nestby` attribute, an element from the list is generated for each tuple from the base view.

There is no limitation regarding the number of lists in each nesting level, nor regarding the number of levels of nesting (except the restrictions imposed on particular implementations).

2.3 Several lists at the same level

As an example of an XML document in which an element has two lists, consider the document below, which is to be obtained from the database in Figure 1.

```
<Departments>
  <Department>
    <Name>Computer Science</Name>
    <DeptProfessors>
      <Professor>Ann</Professor>
      <Professor>John</Professor>
    </DeptProfessors>
    <Courses>
      <Course>
        <Name>Database Systems</Name>
        <CourseProfessors>
          <Professor>Ann</Professor>
          <Professor>John</Professor>
        </CourseProfessors>
      </Course>
      <Course>
        <Name>Compilers</Name>
        <CourseProfessors>
          <Professor>Ann</Professor>
        </CourseProfessors>
      </Course>
    </Courses>
  </Department>
  <Department>
    <Name>Philosophy</Name>
    ...
</Departments>
```

As this query involves two independent relationships (a department with its lecturers, and a department with its courses), it is not efficient to use a single SQL query. This SQL query would have a Cartesian product (for each lecturer from a department it would be shown all of the courses in that department, and vice versa).

The solution chosen in XML/SQL is the use of multiple base views, i.e., the use of multiple SQL queries, one for each independent relationship.

In the example, two base views are necessary, given that there are two lists (`DeptProfessors` and `Courses`) composing the same XML element (`Department`). The `QUERY` clause in that case is as follows:

```
<QUERY>
  <SQL idsql="v3">SELECT d.iddept, d.deptname, p.idprof, p.profname
                    FROM professor p, department d
                    WHERE d.iddept = p.iddept
                    ORDER BY d.iddept
  </SQL>
  <SQL idsql="v4">SELECT d.iddept, c.idcourse, c.coursename,
                    p.idprof, p.profname
                    FROM department d, course c, profcourse pc,
```

```

        professor p
WHERE d.iddept = c.iddept AND
      c.idcourse = pc.idcourse AND
      pc.idprof = p.idprof
ORDER BY d.iddept, c.idcourse

</SQL>
</QUERY>

```

One base view (v3) contains a tuple for each lecturer, together with the information about the department. The same is sorted by department identifier. The other base view (v4) contains a tuple for each lecturer of a course, alongside the course data and the department data. This base view is sorted by department identifier and by course identifier. Two sorting attributes are needed, since there are two nesting levels (Departments have Courses which, in their turn, have CourseProfessors).

We give below the CONSTRUCTOR clause for the referred query:

```

<CONSTRUCTOR>
  <LIST tagname="Departments" idsql="v3" nestby="iddept">
    <SEQUENCE tagname="Department">
      <ATOM tagname="Name" source="deptname"/>
      <LIST tagname="DeptProfessors" idsql="v3">
        <ATOM tagname="Professor" source="profname"/>
      </LIST>
    <LIST tagname="Courses" idsql="v4" nestby="idcourse">
      <SEQUENCE tagname="Course">
        <ATOM tagname="Name" source="coursename"/>
        <LIST tagname="CourseProfessors" idsql="v4">
          <ATOM tagname="Professor" source="profname"/>
        </LIST>
      </SEQUENCE>
    </LIST>
  </SEQUENCE>
</LIST>
</CONSTRUCTOR>

```

When an XML/SQL query is built on more than one base view, it is necessary to specify a criterion with which the results from the two queries will be combined in forming the XML elements. This role is fulfilled by the `nestby` attribute, which not only serves as an attribute for grouping tuples from a base view, but it also serves as criterion for matching lines from one base view with lines from the other.

In the example, an XML element `Department` is formed by elements from both base views (v3 and v4). In that case, the relational attribute `iddept` (specified by the `nestby` attribute) serves as criterion for:

- grouping lines from the base view v3,
- grouping lines from the base view v4, and
- matching every group formed from base view v3 with the groups formed from base view v4.

In the remaining aspects, the query behaves as the previous examples. Differently from those examples, though, there are now two nesting levels, indicated by the nested `LIST` constructor.

3 Concluding remarks and future work

The XML/SQL was conceived as a query language for an access layer to legacy relational databases. This layer was intended for building complex objects needed in a transaction, based on declarative specifications. The motivation is to remove from programs the burden of creating complex objects, normally specified in a procedural fashion, which involve several SQL queries and handling of cursors over those queries.

Therefore, the XML/SQL language was designed taking into considerations the following aspects:

- *Availability of a higher abstraction level*
A software module can use the XML/SQL language to access a database and create complex XML objects. The structure of the built objects is not determined by the structure of the database; it is specified in a declarative fashion.
- *Efficiency in the access to relational sources*
XML/SQL uses SQL to build and extract data from relational sources. Thus, it can profit from the query optimisation available in relational DBMS.
- *Grounding on XML*
Not only are the results obtained in XML format, but also the querying instructions are coded in XML. This simplifies the implementation of wrappers for XML/SQL, as it allows XML parsers to be used for parsing queries.

In the present version, XML/SQL does not generate XML attributes. This limitation can be overcome, e.g. by processing XSL transformations on an XML/SQL result.

We are developing a new version of the XML/SQL language which allows XML documents resulting from queries to be altered and returned to the wrapper for it to map document alterations back to the underlying relational database.

We have implemented a wrapper for relational databases which uses the XML/SQL language.

References

- [Bou00] Ronald Bourret. Xml database products, November 2000. <http://www.rpbouret.com/xml/XMLDatabaseProds.htm>.
- [BPSM98] T. Bray, J Paoli, and C. Sperberg-McQueen. Extensible markup language (xml) 1.0. W3C Recommendation, February 1998. <http://www.w3c.org/TR/1998/REC-xml-19980210>.
- [CRF00] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An xml query language for heterogeneous data sources. In *Lecture Notes in Computer Science, Springer-Verlag*, 2000. <http://www.almaden.ibm.com/cs/people/chamberlin/>.

- [CX00] J. Cheng and J. Xu. Ibm db2 xml extender: An end-to-end solution for storing and retrieving xml documents. ICDE'00 Conference, February 2000.
- [D+98] A. Deutsch et al. Xml-ql: A query language for xml. Submission to the World Wide Web Consortium, Aug 1998.
- [FTS00] M. Fernandez, W. Tan, and D. Suciu. Silkroute: Trading between relations and xml. In *Proceedings of Nineth International World Wide Web Conference*, 2000. <http://www.research.att.com/~mff/files/>.
- [I+98] H. Ishikama et al. Xql: A Query Language for XML Data. In *The Query Languages Workshop*, 1998.
- [IBM00] IBM Corp., San Jose. *XML Extender: Administration and Programming (Version 7)*, 2000.
- [ISR00] Xml from databases: Odbc2xml, 2000. Intelligent System Research, Chicago, USA.
- [Lad01] R. Laddad. Xml apis for databases: Blend the power of xml and databases using custom sax and dom apis, January 2001. <http://www.javaworld.com/javaworld/jw-01-2000/jw-01-dbxml.html>.
- [ORA99] Oracle xml sql utility for java. Oracle Corporation, 1999. http://technet.oracle.com/tech/xml/oracle_xsu/.
- [PGMW95] Y. PAPAKONSTANTINOY, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. *IEEE International Conference on Data Engineering*, pages 251–260, March 1995.
- [Vol99] T. Volker. Making legacy data accessible for xml applications, 1999.
- [Wai99] B. Wait. Using xml in oracle database applications, November 1999. <http://technet.oracle.com/tech/xml/>.

A XML/SQL DTD

The DTD of the XML/SQL language is given below.

```

<!ELEMENT XMLSQL (QUERY,CONSTRUCT)>
<!ATTLIST XMLSQL version CDATA #FIXED "1.0">
<!ELEMENT QUERY (SQL+)>
<!ELEMENT SQL (#PCDATA)>
<!ATTLIST SQL idsql ID #REQUIRED>
<!ELEMENT CONSTRUCT (LIST)>
<!ELEMENT LIST (ATOM|SEQUENCE|LIST)>
<!ATTLIST LIST tagname CDATA #REQUIRED
            idsql IDREF #REQUIRED
            nestby NMTOKENS #IMPLIED>
<!ELEMENT SEQUENCE (ATOM|SEQUENCE|LIST)+>
<!ATTLIST SEQUENCE tagname CDATA #REQUIRED>
<!ELEMENT ATOM EMPTY>
<!ATTLIST ATOM tagname CDATA #REQUIRED

```

```
source CDATA #REQUIRED
mandatory (no|yes) "no">
```