XPERANTO: Publishing Object-Relational Data as XML

Michael Carey[1] Daniela Florescu[2] Zachary Ives[3] Ying Lu[4]
Jayavel Shanmugasundaram[4] Eugene Shekita Subbu Subramanian

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
carey@acm.org, daniela.florescu@inria.fr, zives@cs.washington.edu, luy@cs.wisc.edu, jai@cs.wisc.edu,
shekita@almaden.ibm.com, subbu@us.ibm.com

ABSTRACT

Since its introduction, XML, the eXtended Markup Language, has quickly emerged as the universal format for publishing and exchanging data in the World Wide Web. As a result, data sources, including object-relational databases, are now faced with a new class of users: clients and customers who would like to deal directly with XML data rather than being forced to deal with the data source's particular (e.g., object-relational) schema and query language. The goal of the XPERANTO project at the IBM Almaden Research Center is to serve as a middleware layer that supports the publishing of XML data to this class of users. XPERANTO provides a uniform, XML-based query interface over an object-relational database that allows users to query and (re)structure the contents of the database as XML data, ignoring the underlying SQL tables and query language. In this paper, we give an overview of the XPERANTO system prototype, explaining how it translates XML-based queries into SQL requests, receives and then structures the tabular query results, and finally returns XML documents to the system's users and applications.

Keywords
XML, object-relational database, middleware, views, query. processing

1. INTRODUCTION

Since its introduction, XML, the eXtended Markup Language [2], is quickly emerging as the universal format for publishing and exchanging data over the World Wide Web. In this paper, we will focus on the problem of publishing data in object-relational databases as XML. In the business-to-business e-commerce area, there is a widely recognized need to create XML documents by combining one or more object-relational tables (e.g. creating an XML purchase order by joining a customer with information drawn from other tables). For example, a music store might wish to publish its inventory of used instruments on the web, including each instrument's make, model, condition, price, description, and so on, in order to make this information available to specialized web search engines that help musicians find good deals on used instruments. Further, such a store might provide query access to its inventory in order to support web queries such as "find used 5-string Fender Jazz Bass guitars available for between US $500 and US $900 from stores in the San Francisco Bay area". One approach to meeting the needs of such a music store would be to materialize and publish the store's inventory as XML on its web site on a daily basis. A different approach, and the approach on which we shall focus in this paper, is for the store to provide a virtual XML view of its inventory database (which resides in an existing object-relational DBMS) and to directly support XML queries against this view.

In this paper, we describe our research prototype system for publishing database content as queryable XML views. Our focus is on doing so in a "web-friendly"' manner – more specifically, our assumption is that there is likely to be a growing community of XML web site developers who "live and breathe" XML, and who would prefer to work solely in an XML context. The aim of our project, XPERANTO (Xml Publishing of Entities, Relationships, ANd Typed Objects), is to support this class of developers. To this end, we are developing an XML-centric middleware layer that automatically provides a default XML view of existing databases and an XML query facility with which developers can define new, more desirable XML views. These views can also be queried using the same XML query facility, all without the developers having to learn or write SQL. Internally, of course, XPERANTO translates incoming XML queries into SQL, submits them to the underlying database system, receives the queries' answers, and then translates their results back into XML terms.

A key advantage of the XPERANTO "pure XML" philosophy is that XML can be used to model both relational data and relational meta-data in the same framework. Users can thus query seamlessly over relational data and meta-data using an XML query language. For instance, in a stock database where there are separate tables containing stock quotes for each company, with the table names being the same as the corresponding company names, XPERANTO users can issue an XML query that asks for the names of companies (meta-data) whose stock value (data) exceeded $100 on any day. In this sense, XPERANTO provides a query capability that is more powerful than SQL.

---

[1] Currently at Propel, 2350 Mission College Blvd., Santa Clara, CA 95054.
[2] Work done while the author was visiting the IBM Almaden Research Center from INRIA, Le Chesnay, France.
[3] Work done while the author was visiting the IBM Almaden Research Center from the University of Washington, Seattle, WA 98155
[4] Work done while the author was visiting the IBM Almaden Research Center from the University of Wisconsin, Madison, WI 53706.

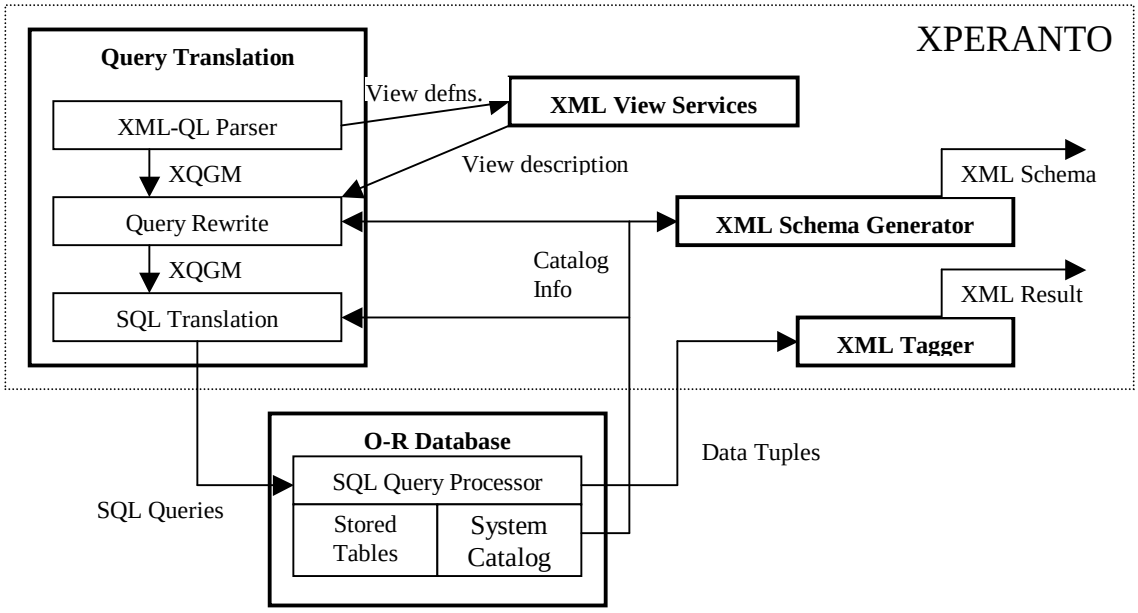

Figure 1: XPERANTO Architecture

The XPERANTO way of publishing object-relational data as XML is unique in many ways. Unlike other similar systems that we are aware of, such as SilkRoute [6], XPERANTO takes the *pure XML, single query language* approach to solving the problem. Thus users and developers of XPERANTO need only be familiar with XML and an XML query language and need not know SQL or learn a new query language (such as RXL [6]). Further, as mentioned above, the pure XML approach adopted by XPERANTO allows for a more powerful query capability because both relational data and meta-data can be represented and queried in the same framework. XPERANTO is also unique in that it publishes not only relational data as XML, but also object-relational structures, including such features as typed tables and columns, oids and references, inheritance, and collections. Finally, unlike [6], XPERANTO pushes all relational logic, such as join and merge, into the object-relational engine thus fully exploiting the sophisticated query processing capability of object-relational databases.

2. XPERANTO ARCHITECTURE

XPERANTO is organized into four major software components, which are further broken down into smaller logical sub-components. As shown in Figure 1, the major components of XPERANTO are: Query Translation, XML View Services, the XML Schema Generator, and the XML Tagger. The core of XPERANTO, and the primary focus of this paper, is the Query Translation component. This component translates from the XML query language used by clients (currently XML-QL [4]) into the appropriate dialect of SQL for the underlying O-R DBMS. The main role played by each of the sub-components in Figure 1 is described below.

- **XML-QL Parser:** Takes an XML-QL query and generates XQGM (XML Query Graph Model) – a language-neutral intermediate representation for XML queries. XQGM shields XPERANTO from the details of a particular XML query language. Thus, XPERANTO can easily adapt to the XML query language standard when one becomes available.

- **Query Rewrite:** Takes the XQGM representation of a query, resolves view references, performs XML view composition, and produces a semantically equivalent XQGM representation of the query. It also consults the database system catalogs in case the user query is over both relational data and meta-data.

- **SQL Translation:** Translates XQGM to SQL statements. This sub-component makes use of (potentially cached) database system catalog information to perform type checking etc.

- **XML View Services:** Serves as a storage and retrieval interface for XML-QL view definitions. When views are defined, they are stored in a dedicated table. They can be later retrieved for view unfolding.

- **XML Schema Generator:** Takes (potentially cached) database catalog information and produces schema information for (user-defined and default) XML views and query results.

- **XML Tagger:** Converts tabular SQL query results into structured XML documents.

1. *Create Table* **book** *AS* (bookID *CHAR(30)*, name *VARCHAR(255)*, publisher *VARCHAR(30))*
2. *Create Table* **publisher** *AS* (name *VARCHAR(30)*, address *VARCHAR(255))*
3. *Create Type* **author_type** *AS* (bookID *CHAR(30)*, first *VARCHAR(30)*, last *VARCHAR(30))*
4. *Create Table* **author** *OF* **author_type** (*REF IS* ssn *USER GENERATED*)

Figure 2: DDL for Example Object-Relational Database

```
<simpleType name="string255" source="string"> <maxLength value="255"/> </simpleType>
<simpleType name="string30" source="string"> <maxLength value="30" /> </simpleType>

<complexType name="bookTupleType">
        <element name="bookID"       type="string30" />
        <element name="name"         type="string255" />
        <element name="publisher"  type="string30" />
</complexType>

<complexType name="bookSetType">
        <element name="bookTuple" type="bookTupleType" maxOccurs="*" />
</complexType>

<element name="book" type="bookSetType" />

<complexType name="author_type">
        <element name="bookID" type="string30" />
        <element name="first"  type="string30" />
        <element name="last"   type="string30" />
</complexType>

<complexType name="authTupleType" source="author_type" derivedBy="extension">
         <attribute name="ssn" type="ID" />
</complexType>

<complexType name="authSetType">
         <element name="authTuple" type="authTupleType" maxOccurs="*" />
</complexType>

<element name="author" type="authSetType" />
```

Figure 3: Schema of Default XML View over Example Object-Relational Database

3. XML SCHEMA MAPPING

As mentioned earlier, one of the goals of XPERANTO is to allow XML developers to publish object-relational data in XML form without having to deal with the database system's native schema or SQL query dialect. XPERANTO achieves this goal by providing a default XML view of the database. Developers can then use this default view to write queries and define more situation-appropriate XML views. The structure of an XML view is described using an XML Schema Specification [10] (XML Schema has been designated to supplant the XML DTD [1], adding important features such as data types, value constraints, inheritance, and foreign key information.) We first briefly introduce object-relational database schemas before describing the construction of default XML views.

3.1 Object-Relational Database Schemas

The schemas of object-relational databases are composed of the usual SQL database primitives (schemas, tables/views, columns, basic built-in data types) augmented with a set of additional primitives (structured types, inheritance, object IDs, references, typed tables/views) that enable database designers to define new data types and complex object structures. (See [3][7] for an overview of the object-relational data definition primitives from a DB2 UDB perspective.) In the interest of space, we will explain these primitives through the use of a single object-relational schema example that incorporates a number of them.

Figure 2 shows the Data Definition Language (DDL) used to define an object-relational schema in SQL99 terms. (Let us assume that these definitions are for a schema named *library* within a database called *books*.) The first DDL statement in Figure 2 defines a *book* table. This is a conventional (SQL92) table having three primitive data type columns – *bookID, name,* and *publisher*. The second DDL statement similarly defines a (SQL92) *publisher* table. The third DDL statement defines a SQL99 structured type – *author_type* . This structured type has three attributes, namely *bookID, first,* and *last* (each a primitive data type). The fourth DDL statement creates an *author* table. Rows of the *author* table are objects of type *author_type.* Each row will contain an object ID column (*ssn* is the name chosen by the database administrator for this column), plus one column for each of the type's attributes.

3.2 Default XML Views

Figure 3 shows a fragment of the XML Schema describing the default XML view of the object-relational schema defined in Figure 2. (The XML Schema definition for the publisher table is omitted in the interest of space). XML Schema uses the *complexType* element to define complex element structures. In the default views produced by XPERANTO, structured types in an object-relational schema are thus directly mapped to the corresponding XML Schema complexType definitions. This mapping is shown in Figure 3, where the DB2 structured type *author_type* has been mapped to a similarly named XML Schema complexType. The XML Schema complexType has sub-elements named *bookID*, *first*, and *last*, corresponding to the attributes of *author_type.* Note that these sub-element types are constrained versions of the basic string type (the XML Schema base type *string*) with maximum lengths specified; we use the XML Schema *simpleType* element to define each type separately.

A conventional (SQL92) table is mapped to a corresponding XML Schema element with the table's name. This element is defined to hold multiple occurrences of another element, namely the table's tuple type element. Thus, in our ongoing example, the XML Schema description corresponding to the *book* table is obtained by first defining a *bookTupleType* whose sub-elements are obtained from the column names of the *book* table (elements *bookID*, *name*, and *publisher*). Then, in order to define the type for the *book* element itself, a *bookSetType* type is defined as being zero or more occurrences of elements of type *bookTupleType.* The book table is then mapped to an element having the name *book* and type *bookSetType.*

Typed (SQL99) tables are handled in a manner similar to that of conventional (SQL92) tables. There are, however, two significant differences. The first difference lies in the use of the XML Schema type extension facility to derive tuple types for typed tables. In our example, the *authorTupleType* is derived from *author_type* by adding an extra sub-element, *ssn*. The second difference lies in the mapping of the object ID columns and object reference columns (if any) that appear in typed tables. XPERANTO uses the ID and IDREF type facilities of XML Schema to map these SQL99 concepts. For example, the *authTupleType* definition has an attribute *ssn* of type ID. Though not shown in this example, XPERANTO can also capture SQL99 type hierarchies using the XML Schema type extension facility.

4. QUERY PROCESSING AND XML DOCUMENT CONSTRUCTION

Once XPERANTO publishes a default XML view of an object-relational database, users can then pose queries and define more complex views using an XML query language. Figure 4 shows an example XML-QL query that selects information about books published by a publisher having a name that contains the string "Wesley". For each book, the query constructs a *book* element having the book name, the publisher of the book (the first nested sub-query), and the authors of the book (the second nested sub-query). The result of the query therefore will be a tree-structured document where each *book* element contains information about the publisher(s) and authors of the book. We will use this example query for the remainder of this paper to illustrate query processing in XPERANTO. We first describe the query rewrites performed for XML view composition before describing SQL query generation and XML document construction.

4.1 XML Query Rewriting

As mentioned earlier, XPERANTO allows users to define complex (virtual) XML views over the default XML view using an XML query language. Other XML views may be defined in terms of these XML views and user queries (in the same XML query language used for view definition) can then be posed over them. In fact, in many cases, end users may never see the default XML view but may only see a more sophisticated, application-specific XML view created by an administrator. The goal of the query rewrite engine is to perform XML view composition and simplify complex user queries over complex XML views and produce equivalent simple queries over the default XML view.

In order to perform XML view composition effectively, XPERANTO translates user queries into an intermediate representation suitable for view composition. This intermediate representation, called XQGM (XML Query Graph Model), closely mirrors the QGM (Query Graph Model) representation used for rewriting queries in the commercial DB2 UDB object-relational database system [8]. There are three main reasons for choosing XQGM as the intermediate representation. First, it ensures that the XPERANTO query rewrite engine will be "upward compatible" with next-generation XML query languages, which will most likely have sophisticated SQL features such as aggregation, null values, universal and existential quantification, etc. [5]. Second, it becomes easier to translate queries in an XML query language to SQL queries because both are represented using similar structures. Finally, the XQGM rule engine can inherit much of the rules and extensibility properties of QGM rule engine, which has proved to be very effective for SQL view composition.

```
WHERE <library.book.bookTuple>
                <bookID> $bid </>
                <name> $bname </>
                <publisher> $bpub </>
          </> IN “db2:xml:books/library”,
          $bpub LIKE “Wesley”
CONSTRUCT <book id=$bid>
                    <name> $bname </>
                    {WHERE <library.publisher.publisherTuple>
                                <name> $bpub </>
                                <address> $addr </>
                           </> IN “db2:xml:books/library”
                     CONSTRUCT <publisher>
                                    <address> $addr </>
                                </>}
                    {WHERE <library.author.authorTuple>
                                <bookID> $bid </>
                                <first> $fname </>
                                <last> $lname </>
                           </> IN “db2:xml:books/library”
                     CONSTRUCT <author first=$fname last=$lname/>}
               </>
```

Figure 4: Example XML-QL Query over Default XML View

It is important to note that using a QGM-like representation does not in any way tie XPERANTO to the DB2 UDB object-relational database system. XPERANTO merely uses an internal representation like QGM in the middleware for the purpose of XML query rewrites and can work on top of any object-relational database system. There are, however, some extensions that need to be made to QGM to make it appropriate for XML query languages. Specifically, means to represent, navigate and construct nested XML elements needs to be added to QGM. XQGM does this by adding an XML type and by supporting new XML-specific functions for navigating (example functions are *GetSubElements*, *GetAttributes*) and constructing (example functions are *CreateElement*, *CreateAttribute*) elements of this type. These functions are modeled in XQGM the same way that SQL functions, such as *max* and *concat*, which operate on SQL types, are modeled in QGM.

The main purpose of XQGM query rewrites is the elimination of unnecessary XML element and attribute construction for those elements that are constructed in intermediate views but do not appear in the final query result. This is done by performing functional composition and exploiting certain equivalences. As an example, consider an XML element created in an XML view. This is represented using the *CreateElement* function in the corresponding XQGM representation of the view. This function takes as its inputs the tag name of the element to be constructed, the list of attributes of the element to be constructed, and the list of sub-elements of the element to be constructed. Now assume that an user query over the view asks for all the sub-elements of the constructed element (and does not require the constructed element to be returned). In this case, it is unnecessary to construct the element during query execution because the sub-elements of the constructed element can be directly returned to the user. In XQGM, obtaining the sub-elements of the constructed element is represented using the *GetSubElements* function and query rewrite exploits the fact that the *CreateElement* function (of the view) and the *GetSubElements* function (of the query) compose to just return the list of sub-elements that are passed as an input parameter to the *CreateElement* function. Unnecessary element creation is thus avoided. Similar functional equivalences are used to handle recursion and wild cards in XML queries.

Special techniques are required when the user queries over both object-relational data and meta-data. For example, consider an XML query over the example default view that asks for the tag names of all the sub-elements of “library” (these tag names represent table names) that contain a sub-element having the tag name (this represents a column name) “name” and having the content (this represents a column value) “Addison-Wesley”. This cannot be translated to a SQL query because SQL does not support seamless querying over data and meta-data. In such cases, during query rewrite, XPERANTO automatically generates SQL queries over the database catalog in order to obtain the relevant meta-data (all the tables having a column named “name”, in our example) and incorporates this information in the XQGM representation of the query. The resulting XQGM representation does not access meta-data information and can be directly translated to SQL. Space constraints preclude a more detailed discussion of query rewrite.

4.2 SQL Generation and XML Document Construction

Once the XPERANTO query rewrite engine performs view composition, the resulting XQGM structure represents a query over the default XML view. The final step in the XPERANTO query translation process is then to create a hierarchical, XML document from flat, relational tables as per the query specification. There are many implementation alternatives to achieve this translation, and XPERANTO uses one of the most

```
WITH OuterUnion (type, bookID, bookName, pubName, pubAddr, authFirst, authLast) AS (
    SELECT '0',  b.bookID, b.name, NULL, NULL, NULL, NULL
    FROM     book b
    WHERE  b.publisher LIKE "Wesley"
UNION ALL
    SELECT '1',  b.bookID, NULL, p.name, p.address, NULL, NULL
    FROM     book b, publisher p
    WHERE  b.publisher LIKE "Wesley" and b.publisher = p.name
UNION ALL
    SELECT '2',  b.bookID, NULL, NULL, NULL, a.first, a.last
    FROM     book b, author a
    WHERE  b.publisher LIKE "Wesley" and b.bookID = a.bookID
)
SELECT * FROM OuterUnion ORDER BY bookID, type
```

Figure 5: Outer Union SQL Query Corresponding to our Example

efficient and robust approaches called the "sorted outer union" approach [9]. In the sorted outer union approach, there are two distinct phases in constructing the result XML document. In the first phase, the (object-relational) data that is necessary to construct the result document is generated. In the second phase, the data is tagged to produce the result XML document. The query processing capabilities of the object-relational engine are used for the first phase, while a tagger in the XPERANTO middleware is used for the second phase. (In cases where there is more XML support in the underlying database system, the tagging can also be done inside the database engine [9].)

Figure 5 shows the SQL query that produces the relational data for the query shown in Figure 4. The SQL query is a union of many sub-queries. Each SQL sub-query corresponds to a (sub)query in the original XML-QL query (see Figure 4). Thus the first sub-query produces book information, the second sub-query produces publisher information and the third sub-query produces author information. Since each sub-query only produces information about one entity, only some of the fields are filled in and the others are null. A special type field (the first field) is added to distinguish the contents produced by each sub-query. The result of the union is sorted by the bookID and type so that in the result XML document, all information about a book appears together, and all publisher information of a book appears before all author information of the same book. Since the information in the result of the query is in document order, a *constant space, streaming* tagger can consume the result and construct the result XML document. More details of the sorted outer union approach, such as optimizations using common sub-expressions etc., can be found in [9].

5. CONCLUSION AND FUTURE WORK

In this paper, we have described a systematic approach to publishing XML data from existing object-relational databases. As we have explained, our work on XPERANTO is based on a "pure XML" philosophy – we are building the system as a middleware layer that makes it possible for XML experts to define XML views of existing databases in XML terms. As a result, XPERANTO makes it possible for its users to create XML documents from object-relational databases without having to deal with their native schemas or SQL query interfaces. XPERANTO also provides a means to seamlessly query over object-relational data and meta-data. Our plans for future work include providing support for insertable and updateable XML views. We are also exploring the construction and querying of XML documents having a recursive structure, such as part hierarchies and bill of material documents.

6. REFERENCES

[1] J. Bosak, T. Bray, D. Connolly, E. Malor, G. Nicol, C.M. Sperberg-McQueen, "W3C XML Specification DTD," http://www.w3.org/XML/1998/06/xmlspec-report.htm.

[2] T. Bray, J. Paoli, C.M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0," http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm.

[3] M.J. Carey, et. al., "O-O, What's Happening to DB2?", Proceedings of the VLDB Conference, Scotland, 1999.

[4] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, "XML-QL: A Query Language for XML," 8th International World Wide Web Conference, Toronto, May 1999.

[5] P. Fankhauser, M. Marchiori, J. Robie, "XML Query Requirements", http://www.w3.org/TR/xmlquery-req.

[6] M. Fernandez, W. Tan, D. Suciu, "SilkRoute: Trading Between Relations and XML," 9th International World Wide Web Conference, Amsterdam, May 2000 (to appear).

[7] Y. C. Fuh, et. al., "Implementation of SQL3 Structured Types with Inheritance and Value Substitutability", Proceedings of the VLDB Conference, Scotland, 1999.

[8] H. Pirahesh, J. M. Hellerstein, W. Hasan, "Extensible/Rule Based Query Rewrite Optimization in Starburst", Proceedings of the SIGMOD Conference, San Diego, 1992.

[9] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, B. Reinwald, "Efficiently Publishing Relational Data as XML Documents", submitted for publication.

[10] H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, "XML Schema Part I: Structures", World Wide Web Consortium (W3C) working draft, http://www.w3.org/TR/xmlschema-1.