XPERANTO: Bridging Relational Technology and XML

Catalina Fan, John Funderburk, Hou-in Lam,
Jerry Kiernan, Eugene Shekita

IBM Almaden Research Center
San Jose, CA 95120

Jayvel Shanmugasundaram

Cornell University
Ithaca, NY 14853

Abstract: XML has emerged as the standard data-exchange format for Internet-based business applications. These applications introduce a new set of data management requirements involving XML. However, for the foreseeable future, a significant amount of business data will continue to be stored in relational database systems. Thus, a bridge is needed to satisfy the requirements of these new XML-based applications while still leveraging relational database technology. This paper describes the design and implementation of the XPERANTO middleware system, which we believe achieves this goal. In particular, XPERANTO provides a general framework to create and query XML views of existing relational data.

1. Introduction

Internet-based business applications promise to dramatically reduce the cost of doing business by providing an automated and secure way to exchange data over the Internet. XML has emerged as the standard data-exchange format for these applications. This has in turn created a new set of data management requirements involving XML. However, for the foreseeable future, most business data will continue to be stored in relational database systems. Thus, a bridge is needed to satisfy the requirements of Internet-based XML applications, while still leveraging relational database technology. This paper describes the architecture of the XPERANTO middleware system, which we believe achieves this goal.

One of the features provided by XPERANTO is the ability to create XML views of existing relational data. XPERANTO does this by automatically mapping the data of the underlying relational database system to a low-level default XML view. Users can then create application-specific XML views on top of the default XML view. These application-specific views are created using XQuery [14], a general-purpose, declarative XML query language currently being standardized by W3C. XPERANTO materializes XML views on demand, and does so efficiently by pushing down most computation to the underlying relational database engine.

Another feature provided by XPERANTO is the ability to query XML views of relational data. This is important because users often desire only a subset of a view's data. Moreover, users often need to synthesize and extract data from multiple views. In XPERANTO, queries are specified using the same language used to specify XML views, namely XQuery. XPERANTO executes queries efficiently by performing XML view composition so that only the desired relational data items are materialized.

In summary, XPERANTO provides a general means to publish and query XML views of existing relational data. Users always use the same declarative XML query language (XQuery) regardless of whether they are creating XML views of relational data or querying those views. The high-level architecture of the XPERANTO system is depicted in Figure 1.

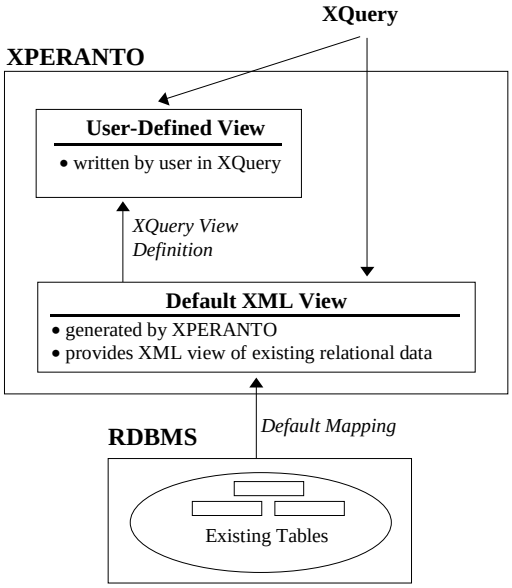

Figure 1: The XPERANTO High-level Architecture

2. Comparison to Related Approaches

There are other approaches that provide solutions to the problems solved by XPERANTO. In this section, we discuss some of those approaches and contrast them to XPERANTO.

XSL-T Processors

XSL-T [15] processors can be used to create and query XML views of relational data. In order to do this, the desired relational data is first materialized in some canonical XML format, and then an XSL-T processor is used to transform the canonical format to the desired XML view. Views over views and queries over views can be processed similarly. XPERANTO differs from this approach in three respects. Firstly, since this approach requires relational data to be materialized before XSL-T processing, selection predicates in queries cannot be pushed down to the relational engine. Thus, unlike XPERANTO, a large amount of unnecessary data may have to be materialized. Also, intermediate XML fragments that do not appear in the final query result may have to be materialized. This is because, unlike XPERANTO, intermediate views have to be materialized. Finally, query processing is less efficient in XSLT than in XPERANTO. This is because XPERANTO pushes most of its computation to a relational database engine, which is likely to be far more powerful than an XSL-T processor.

SilkRoute

SilkRoute [4][5] is a system that allows users to create and query XML views of relational data. In SilkRoute, XML views of relational data are created using a special-purpose query language called RXL. XML views are then queried using another language called XML-QL [6]. Thus, unlike XPERANTO, users need to use different languages for creating and querying views. Further, users need to explicitly straddle the relational and XML models by using the special-purpose RXL language.

Oracle

Oracle's XSQL [8] can also be used to create XML views of relational data. However, queries over views are not supported, nor can views be created on top of views. Oracle also enables XML views of relational data to be created using object-relational technology [1]. Nested structures are specified using objects, which are then mapped to XML. This approach has the same limitations as XSQL, however. Oracle also allows XML documents to be stored, and then queried using SQL with XML extensions. But these extensions are not comparable in power to an XML query language such as XQuery. Moreover, they force application developers to explicitly straddle the relational and XML models. In contrast, XPERANTO, provides a "pure XML" solution for querying XML documents.

SQL Server

Microsoft's SQL Server [7] lets users create XML views of relational data using something called an XDR Schema, which is a proprietary XML schema with relational annotations. Users can query an XDR-generated view, but in contrast to XPERANTO, a different language (XPath [13]) is used for queries than the one used for view definition. Moreover, queries are restricted to Xpath [13] expressions, which cannot support joins. Views on top of views are also not supported.

3. Creating and Querying XML Views of Relational Data

One of the key features provided by XPERANTO is that users can create and query XML views of relational data using a standard XML query language, namely XQuery. We begin this section by describing how user-defined views are created in XPERANTO. We then describe how queries over user-defined views are processed. The main purpose of this section is to help readers understand XPERANTO's overall architecture. Consequently, only a high-level description of view and query processing is provided. For more details on those topics, readers can turn to [11].

3.1 The User's Perspective

As a starting point, XPERANTO automatically creates something referred to as the *default XML view,* which is a low-level XML view of the underlying relational database. Users can then define their own views on top of the default view using XQuery. Moreover, views can be defined on top of views to achieve higher levels of abstraction. The main advantage of this approach is that a standard, general-purpose XML query language is used to create and query views. This is in contrast to approaches such as [2][4][7], where a proprietary language is used to define the initial XML view of the underlying relational database. By using a general-purpose XML query language, XPERANTO gains another important advantage, namely, it allows arbitrarily complex views and queries to be expressed. This is in contrast to approaches such as [2][7], which provide limited querying capabilities without support for joins and recursion.

order

id	custname	custnum
10	Smith Construction	7734
9	Western Builders	7725

item

oid	desc	cost
10	generator	8000
10	backhoe	24000

payment

oid	due	amt
10	1/10/01	20000
10	6/10/01	12000

```
<db>
  <order>
    <row> <id>10 </id> <custname> Smith Construction </custname> <custnum> 7734 </custnum> </row>
    <row> <id> 9 </id> <custname>Western Builders </custname> <custnum> 7725 </custnum> </row>
  </order>
  <item>
     <row> <oid> 10 </oid> <desc> generator </desc> <cost> 8000 </cost> </row>
     <row> <oid> 10 </oid> <desc> backhoe </desc> <cost> 24000 </cost> </row>
  </item>
  <payment>
    … similar to <order> and <item>
  </payment>
</db>
```

Figure 2: A Purchase Order Database and its Default XML View

Figure 2 illustrates what the default view would look like for a simple purchase-order database. As shown, the database consists of three tables, one table to keep track of customer orders, a second table to keep track of the items associated with an order, and a third table to keep track of the payments due for each order. Items and payments are related to orders by an order ID (oid). In the default XML view, top-level elements correspond to tables, with table names appearing as tags. Row elements are nested under these. Within a row element, column names appear as tags and column values appear as text. Although not shown, an XML Schema [12] associated with the default view captures primary- and foreign-key relationships.

Continuing the example, suppose a user wants to publish the purchase-order database as a list of orders in the XML format shown in Figure 3. There, each order appears as a top-level element, with its associated items and payments (ordered by due date) nested under it. To transform the default view into the desired XML format, a user-defined view called “orders” is created, as shown in Figure 4. The view definition is fairly straightforward. An XQuery FLWR expression (lines 2-22) is used to construct each order element. The “for” clause on line 2 causes the variable $order to be bound to each “row” element of the order table. The Xpath [13] expression appearing in line 2 describes how to extract each “row” element from the order table. It basically says to start at the root of the default view, navigate to each “order” element nested under it, and then navigate to each “row” element nested under those “order” elements. The constructor for each new “order” element is given in lines 4-22. For a given order, nested FLWR expressions are used to construct its list of associated items (lines 6-13) and payments (lines 14-21). The predicate on line 8 ($order/id = $item/oid) is used to join an order with its items. Similarly, the predicate on line 16 ($order/id = $payment/oid) is used to join an order with its payments.

```
<order >
  <customer> Smith Construction </customer>
   <items>
     <item> <description> generator </description> <cost> 8000 </cost> </item>
     <item> <description> backhoe </description> <cost> 24000 </cost> </item>
   </items>
   <payments>
     <payment due="1/10/01"> <amount> 20000 </amount> </payment>
     <payment due="6/10/01"> <amount> 12000 </amount> </payment>
   </payments>
</order>
<order>
   <customer> Western Builders </customer>
  …
</order>
```

Figure 3: XML Purchase Order

```
1.  create view orders as (
2.    for $order in view("default")/order/row
3.    return
4.      <order>
5.        <customer> $order/custname </customer>
6.        <items>
7.          for $item in view("default")/item/row
8.          where $order/id = $item/oid
9.          return
10.           <item>
11.             <description> $item/desc </description> <cost> $item/cost </cost>
12.           </item>
13.       </items>
14.       <payments>
15.         for $payment in view("default")/item/row
16.         where $order/id = $payment/oid
17.         return
18.           <payment due=$payment/date>
19.             <amount> $payment/amount </amount>
20.           </payment> sortby(@due)
21.       </payments>
22.     </order>
23. )
```

Figure 4: User-defined XML View

```
1.  for $order in view("orders")
2.  let $items = $order/items
3.  where $order/customer like "Smith%"
4.  return $items
```

Figure 5: Query over user-defined XML View

Once the "orders" view has been created, queries can be issued against it. This is illustrated in Figure 5, which shows a query that extracts a list of "item" elements from the "orders" view for the customer whose name begins with "Smith".

3.2 Query Processing

In many ways, the heart of XPERANTO is its query processor, which executes XQueries over XML views. One of the key features of XPERANTO's query processing architecture is that it can perform view composition, so that only the desired relational data is materialized. This is in contrast to an approach like

XSLT [15], where intermediate views have to be materialized. XPERANTO's query processing architecture also allows it harness the full power of the underlying relational database system by pushing most memory- and data-intensive computation down to the relational engine.

XPERANTO's query processing architecture is shown in Figure 6. An XQuery is initially parsed and converted to an intermediate query representation called the XML Query Graph Model (XQGM). The query is then composed with the XML views it references and rewrite optimizations are performed to eliminate the construction of intermediate XML fragments, unroll recursion, and push down predicates. In the final step, the modified XQGM is processed by the Computation Pushdown module, which separates the XQGM into two parts. The first part captures all the memory- and data-intensive processing, which gets pushed down to the relational database engine as a single SQL query. The second part is a tagger graph structure, which the Tagger Runtime module uses to construct the XQuery result in a single pass over the results of the SQL query. The XQuery result is then returned to the user. The steps in executing an XQuery are depicted by the solid lines in Figure 6.

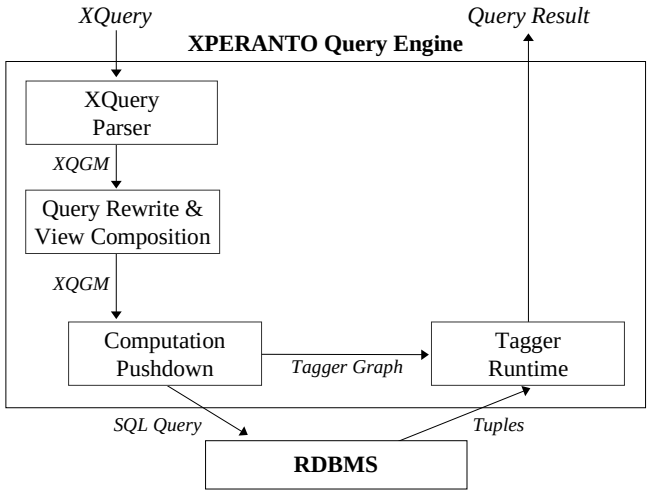

Figure 6: XPERANTO Query Processing Architecture

To provide a concrete example of how an XQuery is processed in XPERANTO, we turn again to the query in Figure 5, which extracts a list of "item" elements from the "orders" view for the customer whose name begins with "Smith". The query is first parsed and converted to the XQGM shown in Figure 7. The easiest way to read the XQGM is bottom-up. First, the fact that the query is over the "orders" view is represented using a view operation (box 1), with the variable $order bound each "order" element produced by the view. The next box selects the order whose "customer" element begins with "Smith", and the final box projects out the list of "item" elements of that order.

As shown, XQGM is a fairly high-level intermediate representation. It was designed with flexibility in mind, so it can be easily adapted as the XQuery standard solidifies. Since the goal is to push down as much computation as possible to the relational engine, we purposely chose an intermediate representation

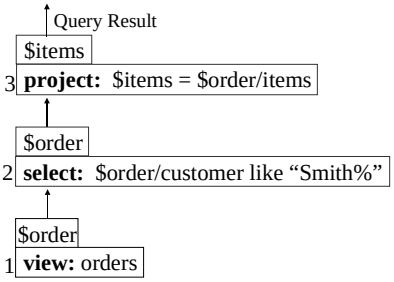

Figure 7: The XQGM for a user Query

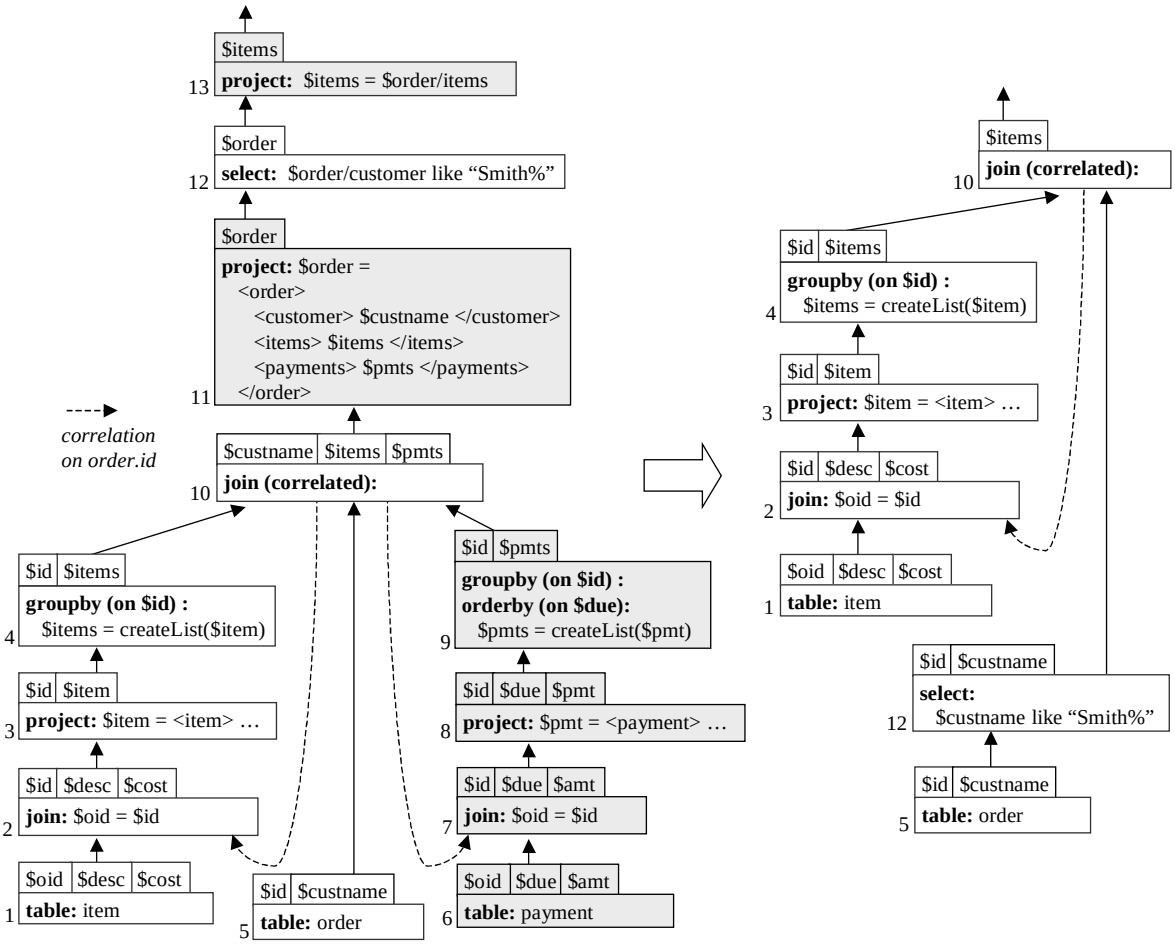

Figure 8: XML View Composition

that can be easily translated to SQL. Note that XQGM borrows from other intermediate representations for XML [3] as well as SQL [9].

After a query is converted to XQGM, it gets composed with the view it references (in this case “orders”) to produce an equivalent XQGM that queries directly over relational tables. This makes it possible to avoid materializing intermediate views. To eliminate view references, XPERANTO first creates an XQGM for each referenced view by parsing the XQuery used to define it. The resulting XQGM graphs are then grafted in place of their view references. Finally, the modified XQGM is rewritten to produce an XQGM that operates directly over relational tables.

Turning back to our example, Figure 8 illustrates the result of grafting the XQGM for the "orders" view in place of its reference. What was box 1 in Figure 7 has now been replaced by boxes 1-11 in Figure 8. Focusing on the left side of Figure 8 and working bottom-up, the correlation on order.id (box 10) is used to drive the construction of "item" and "payment" elements (boxes 1-4 and boxes 6-9, respectively). Next, related orders, items, and payments are joined (box 10). Then each customer name ($custname), list of "item" elements ($items), and list of payment elements ($pmts) output by the join is used to construct an "order" element (box 11). The remainder of the XQGM (boxes12-13) is the same as before.

The right side of Figure 8 shows what the modified XQGM looks like after being rewritten. By looking at the Xpath expression in the top-most project (box 13) and how it maps to the XML constructed for each "order" element (box 11), the query processor is able to determine that the shaded boxes can be eliminated (boxes 6-9, and 11). This is because the XML they construct ("payment" elements) do not appear in the query result. Finally, the query processor is able to determine that the predicate on customer (box 12) ultimately maps to the custname column of the order table, so it can be pushed down. Although it is not shown, the join in Figure 8 is decorrelated before any SQL is generated.

Recall, that in the final steps of query processing, the Computation Pushdown module separates the XQGM into two parts. The first part captures all the memory- and data-intensive processing, which is pushed down to the relational database engine as a single SQL query. The second part is a tagger graph structure, which the Tagger Runtime module uses to construct the XQuery result from the results of the SQL query. Space limitations prevent us from discussing how these final steps in query processing are carried out. More details can be found in [11]. We will show the SQL query produced, however.

The SQL query for our example appears in Figure 9. XPERANTO uses the "sorted outer union" technique describe in [10], which has been shown to be one of the most efficient and stable strategies for materializing relational data for the purpose of constructing XML documents. As shown, the union has two inputs, one for the order table, and the other for the item table. The result of the union is sorted by order ID, type (0=order, 1=item), and due date to put the data in what amounts to document order. This enables the Tagger Runtime to construct the output XML in a single pass over the SQL result.

```
1.  select type, oid, custname, desc, cost
2.  from (select 0, order.id, order.custname, null, null
3.          from order
4.          where order.custname like "Smith%"
5.          union all
6.          select 1, order.id, null, item.desc, item.cost
7.          from order, item
8.          where order.id = item.oid
9.         ) as  (type, oid, custname, desc, cost)
10. order by oid, type, due
```

Figure 9: Sorted Outer Union SQL Query

Before concluding, it is important to note that, although our example query is over a single XML view, in practice an XQuery can span multiple views in XPERANTO. This is an important capability because users often need to synthesize and extract data from multiple views.

4. Conclusions

XML-based applications are imposing new requirements on data management systems. These requirements include publishing and querying existing relational data as XML. This paper has described the design and implementation of the XPERANTO middleware system that harnesses relational database technology to meet these requirements. In particular, XPERANTO exposes relational data as an XML view. Users can then query these XML views using a general-purpose, declarative XML query language (XQuery), and they can use the same query language to create other XML views. Thus, users of the system always work with a single query language In addition to providing users with a powerful system that is simple to use, the declarative nature of user queries allows XPERANTO to perform optimizations such as view composition and pushing computation down to the underlying relational database system.

References

[1] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, R. Murthy, "Oracle8i – The XML Enabled Data Management System", Proceedings of the ICDE Conference, California, USA, March 2000.

[2] J. M. Cheng, J. Xu, "XML and DB2", Proceedings of the International Conference on Data Engineering, California, USA, March 2000.

[3] V. Christophides, S. Cluet, J. Simeon, "On Wrapping Query Languages and Efficient XML Integration", Proceedings of the SIGMOD Conference, Dallas, Texas, June 2000.

[4] M. Fernandez, W. Tan, D. Suciu, "SilkRoute: Trading Between Relations and XML", Proceedings of the World Wide Web Conference, Toronto, Canada, May 1999.

[5] M. Fernandez, A. Morishima, D. Suciu, "Efficient Evaluation of XML Middle-ware Queries", Proceeding of the SIGMOD Conference, Santa Barbara, California, May 2001 (to appear).

[6] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, "XML-QL: A Query Language for XML", Proceedings of the World Wide Web Conference, Toronto, Canada, May 1999.

[7] Microsoft Corporation, http://www.microsoft.com/xml.

[8] Oracle Corporation, http://technet.oracle.com/tech/xml.

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

[10] J. Shanmugasundaram, et. al., "Efficiently Publishing Relational Data as XML Documents", Proceedings of the VLDB Conference, Cairo, Egypt, September 2000.

[11] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, J. Funderburk, "Querying XML Views of Relational Data", submitted for publication to VLDB 2001.

[12] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation, October 2000. See http://www.w3c.org/TR/REC-xml.

[13] World Wide Web Consortium, "XML Path Language (XPath) Version 1.0", W3C Recommendation, November 1999. See http://www.w3c.org/TR/xpath.html.

[14] World Wide Web Consortium, "XQuery: A Query Language for XML", W3C Working Draft, February 2000. See http://www.w3c.org/TR/xquery.

[15] World Wide Web Consortium, "XSL Transformation (XSLT) Version 1.0", W3C Recommendation, November 1999. See http://www.w3c.org/TR/xslt.html.