Querying XML Views of Relational Data

Jayavel Shanmugasundaram Jerry Kiernan Eugene Shekita Catalina Fan John Funderburk

IBM Almaden Research Center
San Jose, California 95120, USA
{shanmuga, kiernan, shekita}@almaden.ibm.com, {fancy, jfund}@us.ibm.com

Abstract

XML has emerged as the standard data exchange format for Internet-based business applications. This has created the need to publish existing business data, stored in relational databases, as XML. A general way to publish relational data as XML is to provide XML views over relational data, and allow business partners to query these views using an XML query language. In this paper, we address the problem of evaluating XML queries over XML views of relational data. This paper makes two main contributions. The first is a general framework for processing arbitrarily complex queries specified using the XQuery query language. The second is a technique for efficiently evaluating XML queries by pushing most of the query computation down to the relational engine.

1. Introduction

XML has emerged as the standard data exchange format for Internet-based business applications. This has created the need to publish existing business data as XML. Since most business data is currently stored in relational database systems, the problem of publishing relational data as XML assumes special significance. A general and flexible way to publish relational data as XML is to create (possibly many) XML views of the underlying relational data. Each of these XML views can provide an alternative, application-specific view of the underlying relational data. Through these XML views, business partners (and other XML application developers) can access existing relational data as though it was in some industry-standard XML format.

Once XML views are created over relational data, the next question that arises is how business partners and XML application developers are to use these views. One simple solution is to materialize the entire XML view on request and return the resulting XML document. The main problem with this approach is that, in many cases, applications do not require the whole view to be materialized. For example, in an XML view of available items, a business partner may only be interested in a particular item. Materializing the availability of all the items would be wasteful in this case because it would result in unnecessary computation. A better solution is to support queries over XML views so that business partners can retrieve only the data items of interest. Supporting queries over XML views also allows application developers to synthesize data from different XML views.

In this paper, we address the problem of evaluating XML queries over XML views of relational data. We consider the case where views and queries are specified using XQuery [16], the XML query language currently being standardized by the World Wide Web Consortium. We focus on two issues. The first is the design of a general framework for processing arbitrarily complex XQuery queries, including queries with features such as nested expressions and nested order.

The other area of focus is performance, whereby we present techniques for efficiently evaluating XQuery queries over XML views of relational data. One such technique is XML view composition, which eliminates the construction of all intermediate XML fragments that do not appear in the final query result. Another performance-enhancing technique is what we call "computation push-down". This pushes all data and memory intensive computation in a XQuery query down to the relational engine. As a result, the query processing power of a relational engine is used to efficiently evaluate XML queries. Only a small memory-efficient tagger is required outside the relational engine to tag the SQL results and produce the resulting XML.

We have implemented the techniques described above in the context of the XPERANTO middleware system [3][12], which works on top of any relational database system. During the course of our implementation, we have identified some of the limitations that arise from using a relational query processor for executing XML queries. These limitations are due to the semantic mismatch between XQuery and SQL. In the conclusion of this paper, we identify the causes for this mismatch and propose possible solutions to help overcome this problem.

To summarize, the contributions of this paper are: (a) a general framework for processing XQuery queries with features such as nested expressions and order, (b) a view composition mechanism that eliminates the construction of all intermediate XML fragments, (c) a computation push down mechanism that pushes all data and memory intensive computation in a XQuery query down to SQL, (d) a description of extensions that can enable a relational engine to handle a larger class of XQuery queries.

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

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 1: A Purchase Order Database and its Default XML View

The rest of this paper is organized as follows. Section 2 discusses related work and Section 3 describes our high-level query processing architecture. The next three sections describe the query processing components such as the parser, the view composition module and the computation pushdown module. Section 7 discusses performance aspects, and Section 8 concludes the paper.

2. Related Work

Most major commercial database systems provide a way to create materialized XML views of relational data [1][4][10]. However, in contrast to the approach presented in this paper, most of these systems do not support queries over XML views [1][4]. Microsoft's SQL Server is the only one that supports queries over XML views, but this query support is very limited. This is because queries are specified using XPath [15], which is a subset of XQuery. For instance, unlike XQuery, XPath cannot specify joins.

XML-based data integration systems [2][5] also wrap relational data sources as XML views. In this context, there has been work done on pushing part of XML query execution down to the relational sources, with the remainder of the query being executed by an XML run-time engine in the integration layer [5]. In contrast to this approach, we support a general XML query capability over relational databases without the need for a full-blown XML run-time engine. Our approach also differs in that we generate query plans that are optimized for relational databases. This is important because the choice of query plans can make a significant difference in performance, especially when constructing complex XML results [6][11]. These differences apart, our system can indeed be used as an efficient XML wrapper for relational sources.

SilkRoute [6][7] is a related system that supports queries over XML views of relational data. Our work differs from SilkRoute in the following respects. Firstly, we support a more powerful query facility based on XQuery. XQuery has features like arbitrarily nested expressions and nested order, which are not supported in SilkRoute. Secondly, we use a view composition technique that is complete and yet produces minimal SQL queries. This is in contrast to SilkRoute's view composition mechanism, which produces SQL queries with redundant predicates and joins (these queries can be minimized, but this problem is NP-complete [7]). Thirdly, our computation pushdown mechanism is more general because it takes into account the greater flexibility of XQuery. Finally, unlike SilkRoute, our internal XML query model naturally extends the relational model. Our work can thus serve as the basis for extending relational databases with XML features (see conclusion for details).

3. Query Processing Architecture

In this section, we present our high-level query-processing architecture. We begin by illustrating how XML views are created and queried in XPERANTO.

As a starting point, XPERANTO automatically creates a *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 XML query language is used to create and query views. This is in contrast to approaches such as [4][7][10], where a proprietary language is used to define the initial XML view of the underlying relational database.

Figure 1 shows the default XML view 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 (there is no specific ordering among the

```
<order id="10">
  <customer> Smith Construction </customer>
  <items>
    <item description="generator" >
      <cost> 8000 </cost>
    </item>
    <item description="backhoe">
      <cost> 24000 </cost> </item>
    </item>
  </items>
  <payments>
    <payment due="1/10/01">
      <amount> 20000 </amount>
    </payment>
    <payment due="6/10/01">
      <amount> 12000 </amount>
    </payment>
  </payments>
</order>
<order id="9" >
  …
</order>
```

Figure 2: XML Purchase Order

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

Figure 4: Query over user-defined XML View

table elements). Row elements are nested under the table elements. Within a row element, column names appear as tags and column values appear as text.

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 2. 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 3. An XQuery FLWR expression (short for For-Let-Where-Return expression) in lines 2-23 is used to construct each order element. The "for" clause on line 2 causes the variable $order to be bound to each "row" element of the order table. The XPath [15] expression appearing in line 2 describes how to extract each "row" element from the order table. It basically says to start at the root of the default view, navigate to each "order" element nested under it, and then navigate to each "row" element nested under those "order" elements. The constructor for each new "order" element is given in lines 4-23. For a given order, nested FLWR expressions are used to construct its list of associated items (lines 7-12) and payments (lines 15-21). The predicate on line 8 ($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. The payments are ordered by their due dates (line 21).

Once the "orders" view has been created, queries can be issued against it. This is illustrated in Figure 4, which

```
01. create view orders as (
02.   for $order in view("default")/order/row
03.   return
04.     <order id=$order/id>
05.       <customer> $order/custname </customer>
06.       <items>
07.         for $item in view("default")/item/row
08.         where $order/id = $item/oid
09.         return
10.           <item description=$item/desc >
11.             <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>
21.         sortby(@due)
22.       </payments>
23.     </order>)
```

Figure 3: User-defined XML View

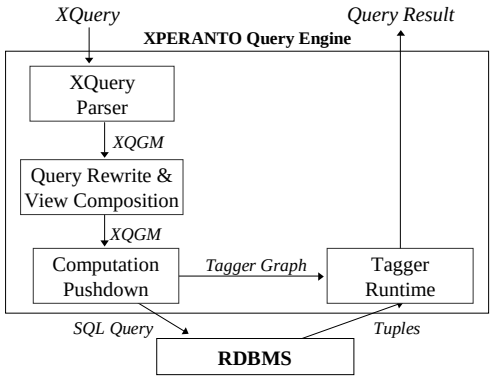

Figure 5: Query Processing Architecture

shows a query that extracts all orders for the customer whose name begins with "Smith".

We now describe our query-processing architecture, which is shown in Figure 5. When a XQuery query is issued over XML views, it is first parsed and converted to an internal query representation called XML Query Graph Model (XQGM). The query is then composed with the views it references and rewrite optimizations are performed to eliminate the construction of intermediate XML fragments and push down predicates. The modified XQGM is then processed by the Computation Pushdown module, which separates the XQGM into two parts. The first part captures all the memory and data intensive processing and 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 XML query result. This is done in a single pass over the row streams resulting from the SQL query. In the remainder of this paper, we describe the major query processing components in detail.

4. Query Parsing

In this section, we present the first phase of query processing corresponding to the XQuery Parser. We first describe XQGM and then present a general approach for converting XQuery queries into XQGM.

4.1. XML Query Graph Model (XQGM)

An intermediate query representation for processing queries over XML views of relational data needs to (a) be powerful enough to capture the full generality of a sophisticated query language such as XQuery, and (b) be amenable to a translation to SQL. We have designed the XQGM query representation with these in mind. XQGM is a natural extension of a SQL internal query representation called Query Graph Model (QGM) [8], which is used in a commercial relational database system. By building upon QGM in this manner, XQGM allows for a natural translation of XQuery queries to SQL. It also enables us to take the vast body of knowledge on relational query optimization and apply it to the XML query problem. In designing XQGM, we have also borrowed from the work on XML query algebras [5][14].

XQGM consists of a set of operators and functions that are designed to capture the semantics of an XML query. Table 1 shows the operators used in XQGM. As can be seen, the operators are a super-set of traditional relational operators. The select, project, join, group by, order by and union operators have the same semantics as their relational counterparts. The project operator is used to invoke functions (described later) in addition to projecting relational results. The table and view operators are used to refer to relational tables and XML view definitions respectively. The unnest operator is used to unnest XML lists. The function operator is used to invoke XQuery valued functions represented in XQGM.

The creation and manipulation of XML objects is done using XML functions. Table 2 presents a list of XML functions and also indicates the operators in which these functions can appear. We now show how XQuery queries can be captured using the XQGM representation. We first provide illustrative examples, and then outline the general principles involved in the translation.

OPERATOR	DESCRIPTION
Table	Represents a table in a relational database
Project	Computes results based on its input
Select	Restricts its input
Join	Joins two or more inputs
Groupby	Applies aggregate functions and grouping
Orderby	Sorts input based on column values
Union	Unions two or more inputs
Unnest	Applies super-scalar functions to input
View	Represents a view
Function	Represents an XQuery function

Table 1: XQGM Operators

The XQGM graph for the view query given in Figure 3 is shown in Figure 6. Recall that the query produces a list of order XML elements, each of which has items and payments nested under it. We first explain the graph at a high level and then provide additional details.

Box 1 represents the *order* table with the two columns that are referenced in the query. Box 10 uses the notion of *correlated joins* (a join operator whose inputs are correlated sub-queries) to compute the list of items and payments associated with each order. Box 11 produces the order XML elements by tagging its inputs. While this tagging is shown as a template in box 11 for illustrative purposes, it is actually implemented as a call to the *cr8Elem* function as shown in Figure 7.

As shown in Figure 7, an *order* element has one attribute, *id*, created using the *cr8Att* function. In addition, an *order* element has three sub-elements, which are *customer*, *items* and *payments*. The *customer* element has no attributes but has a data value fragment represented by the input variable *$custname*. The *items* element again has no attributes but has a list of *item* sub-elements. This list of *item* sub-elements is represented by the input variable *$items*. The *payments* element is created similarly.

Returning to Figure 6, the correlated sub-query represented by boxes 6 through 9 computes the list of payments associated with each order. The payment rows from the *payment* table (box 6) that belong to an order are selected using a correlated predicate on the order id (box 7). Box 8 creates payment elements using the *cr8Elem* function and box 9 aggregates all the payment elements

	XML FUNCTION	DESCRIPTION	OPERATORS
1	cr8Elem(Tag, Atts, Clist)	Creates an element with tag name Tag, attribute list Atts, and contents Clist	Project
2	cr8AttList(A_1, …, A_n)	Creates a list of attributes from the attributes passed as parameters	Project
3	cr8Att(Name, Val)	Creates an attribute with name Name and value Val	Project
4	cr8XMLFragList(C_1, …, C_n)	Creates an XML fragment list from the content (element/text) parameters	Project
5	aggXMLFrags(C)	Aggregate function that creates an XML fragment list from content inputs	Groupby
6	getTagName(Elem)	Returns the element name of Elem	Project, Select
7	getAttributes(Elem)	Returns the list of attributes of Elem	Project, Select
8	getContents(Elem)	Returns the XML fragment list of contents (elements/text) of Elem	Project, Select
9	getAttName(Att)	Returns the name of attribute Att	Project, Select
10	getAttValue(Att)	Returns the value of attribute Att	Project, Select
11	isElement(E)	Returns true if E is an element, returns false otherwise	Select
12	isText(T)	Returns true if T is text, returns false otherwise	Select
13	unnest(List)	Superscalar function that unnests a list	Unnest

Table 2: XML Functions and the operators in which they can appear

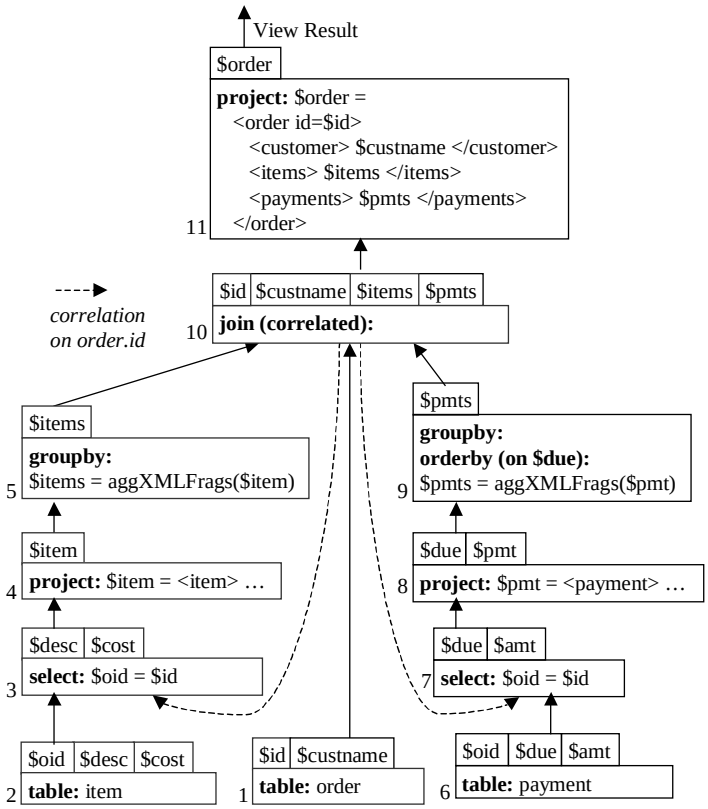

Figure 6: XQGM for the Order XML View

```
cr8Elem(order,
        cr8AttList(cr8Att(id, $id)),
        cr8XMLFragList(cr8Elem(customer,
                               cr8AttList(),
                               cr8XMLFragList($custname)),
                       cr8Elem(items,
                               cr8AttList(),
                               cr8XMLFragList($items)),
                       cr8Elem(payments,
                               cr8AttList(),
                               cr8XMLFragList($pmts))
                      )
       )
```

Figure 7: Expansion of Box 11 in Figure 6

associated with an order into a list. For such grouping operations, the operator can be adorned with an ordering condition that specifies the ordering of elements within the list. In our example, payments of an order are sorted by their due date. The computation of the list of items associated with each order (boxes 2-5) is similar to the computation of payments described above.

As another example of XQGM translation, Figure 8 shows the XQGM representation of the query shown in Figure 4. Box 1 captures the *orders* view referenced in the query. The where clause of the FLWR expression is computed as a correlated sub-query (boxes 2-7). This is done for the following reasons. Firstly, the where clause in XQuery can in general be an arbitrarily complex expression. Therefore, representing the where clause as a correlated query helps set up a separate context for its

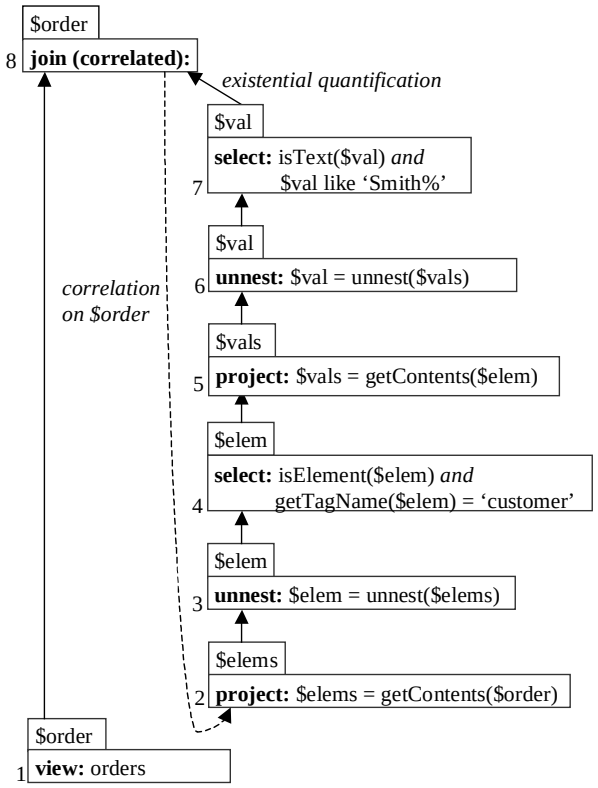

Figure 8: XQGM for Query over Order XML View

computation without inadvertently changing the cardinality or other properties of the main query. Secondly, the where clause in XQuery has implicit existential semantics. In our example, the predicate ($order/customer/text() like 'Smith%') returns true if *any* customer element under order satisfies the predicate (if there exists more than one customer element). By representing the predicate as a correlated sub-query, the results can be existentially quantified (above box 7).

The XQGM representation of the predicate is fairly straightforward. Box 2 gets the contents of a correlated order and box 3 unnests these contents. Box 4 selects only those contents that are sub-elements with tag name 'customer'. Box 5 gets the contents of these elements, box 6 unnests these contents, and box 7 selects the text content whose value satisfies the desired predicate.

XQuery also supports other complex expressions besides FLWR and sortby expressions. These include "let-eval", "if-then-else"' and "quantified" expressions. These expressions have clauses which themselves can be complex expressions. This generality leads to a powerful querying capability but increases the complexity of generating a semantically correct internal query representation. In our system, complex XQuery expressions are represented as correlated sub-queries with their separate context, much like how the where clause of the FLWR expression is represented in Figure 8. Sections 5 and 6 describe how this representation can be simplified using decorrelation and other query rewrite transformations.

5. View Composition

XML views with nested sub-elements are computed from flat relational tables. Navigational operations expressed as path expressions in XQuery queries traverse these nested structures to extract sub-elements and their attributes. Therefore, the query operators that traverse nested structures effectively invert the query operators that create them in a view. Navigational operations can thus be eliminated by undoing the construction of the corresponding elements. Our view composition module performs this query simplification.

Removing all XML navigation operations offers several performance benefits. The obvious benefit is that the construction of intermediate XML fragments – those that do not appear in the final query result – is undone. Thus, only the desired XML fragments are materialized. As we shall shortly see, the other benefit of removing XML navigation functions is that it enables predicates and joins to be pushed down to the relational engine. As a result, there is no need for a full-fledged XML query-processor in the middleware layer. Only a space-efficient tagger, which will be described in more detail in Section 6, is required.

We have developed a complete set of composition rules involving XQGM functions that can be used to remove all XML navigation operations. We first present these rules and then discuss other query rewrite transformations that complement view composition.

5.1. Composition Rules

Functions 6 through 13 in Table 2 represent the functions that capture all the navigation operations in an XQuery query. Table 3 defines twelve composition rules that can be used to eliminate all occurrences of these navigational functions. These rules are complete in the sense that they specify how *all* occurrences of navigational functions can be eliminated. This is done by specifying a composition rule *for every possible input* to a navigational function, which specifies how the navigational function is to be removed for the given input. We now describe the composition rules in detail.

Rule 1 in Table 3 says that the *getTagName* function when applied to the *cr8Elem* can be reduced to the first argument of the *cr8Elem* function (which is the tag name of the created element). Rules 2-5 are defined in a similar manner. Rules 6 and 7 replace the *isElement* function by *true* or *false*, depending on whether the input is an element or not. Rules 8 and 9 are defined similarly. Rule 10 composes the *unnest* function with the *aggXMLFrags* function by simply returning the input to *aggXMLFrags* without performing any aggregation. Rule 11 composes the *unnest* function with the *cr8XMLFragList* function by reducing the *unnest* function to a union of all the arguments of the *cr8XMLFragList* function. Rule 12 is defined similarly.

	FUNCTION	COMPOSES WITH	REDUCTION
1	getTagName	cr8Elem(Tag, Atts, Clist)	Tag
2	getAttributes	cr8Elem(Tag, Atts, Clist)	Atts
3	getContents	cr8Elem(Tag, Atts, Clist)	Clist
4	getAttName	cr8Att(Name, Val)	Name
5	getAttValue	cr8Att(Name, Val)	Val
6	isElement	cr8Elem(Tag, Atts, Clist)	True
7	isElement	Other than cr8Elem	False
8	isText	PCDATA	True
9	isText	Other than PCDATA	False
10	unnest	aggXMLFrags(C)	C
11	unnest	cr8XMLFragList(C_1, …, C_n)	$C_1 \cup \ldots \cup C_n$
12	unnest	cr8AttList(A_1, …, A_n)	$A_1 \cup \ldots \cup A_n$

Table 3: Composition Rules

5.2. Applying Composition Rules

We eliminate all navigational operations by repeated application of the composition rules in Table 3. This step is complemented by a number of other query rewrite transformations that push down predicates, and remove unreferenced columns and operators. We illustrate this step in view composition using an example.

Figure 9 shows the result of composing the query graph in Figure 8 with the view graph in Figure 6. As can be seen, all XML navigation functions in Figure 8 have been composed with their counterparts in box 11 of Figure 6 (box 11 is also expanded in Figure 7). As a result, the selection predicate is specified directly over *$custname* (box 12 in Figure 9). Once navigational functions have been removed, standard query rewrite transformations such as predicate pushdown can be applied. In our example, the predicate has been pushed down to a select box immediately above the *order* table.

Although not illustrated by the above example, all intermediate XML fragments are also removed at this stage of query processing. For example, if the query in Figure 4 had just selected the items in the desired orders, only the XML construction functions that produce items would be present in the output. Since the order elements as a whole would no longer be referenced, they would have been removed from the query graph and hence, would not have been materialized.

6. Computation Pushdown

The goal in this phase of query processing is to push all data and memory intensive operations down to the relational engine as an efficient SQL query. We describe two query processing techniques that make this possible.

6.1. Query Decorrelation

In Section 4, we showed how complex expressions in XQuery are represented using correlations. However, it has been shown in earlier work that executing XML queries as correlated queries over a relational database leads to poor performance [11]. We thus present query decorrelation [13] as a necessary step for efficient XML query execution.

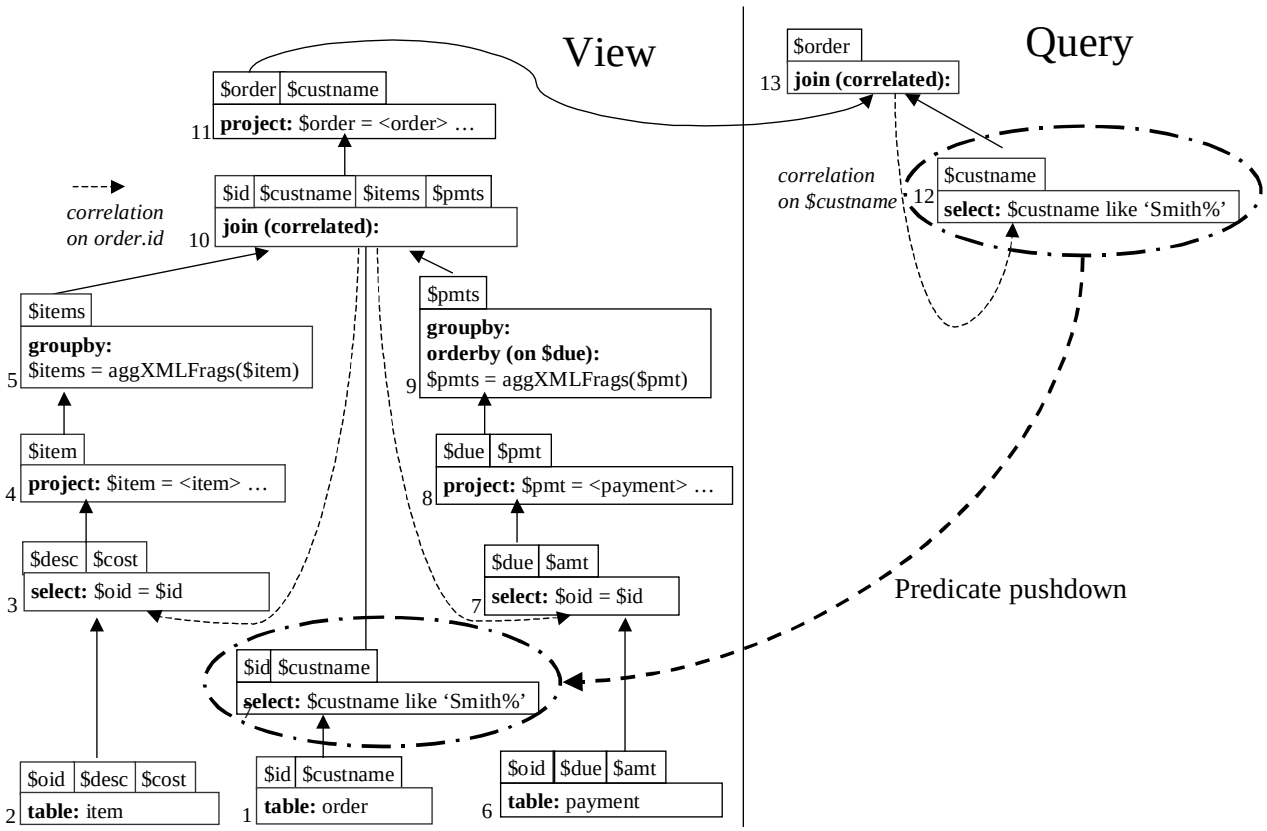

Figure 9: XQGM after View Composition

The result of decorrelating the XQGM of Figure 9 is shown in Figure 10. As shown, the construction of the list of items (boxes 2-5) associated with orders has been decorrelated. This is done by directly joining the selected orders (box 10) with the item table, instead of performing a correlated selection condition. The items are then tagged (box 4) and grouped on the id of the order (box 5) to create a list of items for each selected order. The list of payments associated with each purchase order is computed similarly (boxes 6-9).

Once the list of items associated with orders is computed, these are outer joined with the selected orders (box 11). An outer join is necessary to preserve all selected orders because the join in box 3 would have eliminated orders without any items. This outer joined result is similarly outer joined with payments (box 12).

6.2. Tagger Pull-up

After query decorrelation, the next step is to generate a SQL query that can efficiently produce the relational content for constructing the result XML document. It has been shown in earlier work [11] that the “sorted outer union” SQL query is one of the most efficient and stable techniques for this purpose. However, the generation of the sorted outer union SQL query from the XQGM graph is complicated by the fact that the “tagger operations”, which construct XML fragments, can be mixed with the “SQL operations”, which join or otherwise manipulate relational content (see Figure 10).

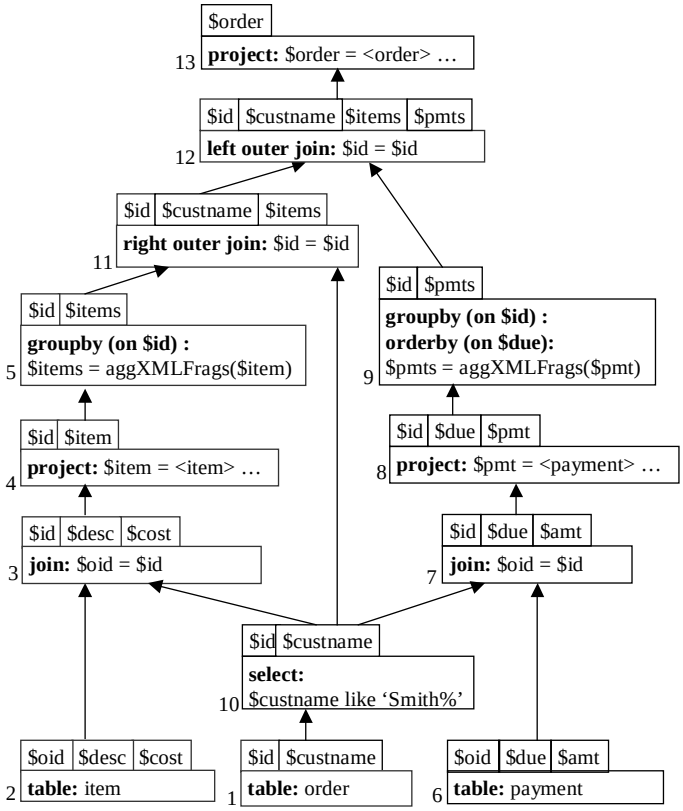

Figure 10: XQGM after Decorrelation

OPERATOR	USAGE	FUNCTIONS
Merge	Merges one or more ordered streams	cr8Elem, cr8Att, cr8XMLFragList, cr8AttList
Aggregate	Computes aggregates	aggXMLFrags
Union	Unions ordered streams	
Input	Manages relational rows	

Table 4: Summary of Tagger Operators

As a result, the tagger and SQL operations need to be separated before the sorted outer union SQL query can be generated. This process of separation is what we call "tagger pull-up".

During tagger pull-up, relational operations are pushed to the bottom of the graph, and XML construction functions are pulled up to the top of the query graph. The bottom portion of the query graph is then converted to a sorted outer union SQL statement and sent to the relational engine for execution. The top portion is transformed into a "tagger run-time" graph, which produces the result XML documents.

The tagger run-time graph consists of a set of tagger operators. In contrast to the relational operators, which are designed for execution by a sophisticated relational engine, the tagger operators are designed for efficient in-memory processing in the middleware. The tagger operators process ordered streams of rows and produce the XML result in constant space, and in a single pass over the SQL query results. The list of tagger operators, along with the functionality of each operator, is shown in Table 4. The XML construction functions that can be implemented in each tagger operator are also shown.

Our system implements a general tagger pull-up algorithm that works for complex query graphs, even recursive ones. There are, however, some queries for which all data and memory intensive computation cannot be pushed down to the relational engine. More details regarding this are presented in Section 8, along with a discussion of how this limitation could be removed.

6.2.1. An Illustrative Example

An example of tagger pull-up is shown in Figure 11. This graph is the result of applying tagger pull-up transformations to the query graph in Figure 10. As can be seen in Figure 11, the bottom part consists of SQL statements, while the top part consists of tagger operators. For the rest of this section, we describe how the tagger operators produce the XML result in a single pass over the SQL results. Details on separating the SQL part from the tagger part are deferred until the next section.

The bottom part of Figure 11 produces three SQL streams. The middle stream produces the selected orders, ordered by their id. The first and third streams produce the desired items and payments, respectively, ordered by the id of the order they are associated with. The third stream is also ordered by the due date to capture the ordering of payments. Since all the results are ordered using the order id, the result XML elements can be constructed in a single pass over the SQL results. This is done by the tagger operators, which work as follows.

When an order flows up through the tagger input operator (box 1), only the items and payments corresponding to that order are let through by their corresponding tagger input operators (boxes 2 and 5). These items and payments are then tagged (boxes 3 and 6) and grouped (boxes 4 and 7). The top tagger operation (box 8) then merges these together to produce the result. This process is repeated for the next order.

Although Figure 11 shows three SQL statements, these are actually executed as a single SQL statement by performing an outer union (an "outer union" can be used to union inputs with different column types). This is done using the efficient sorted outer union technique proposed and evaluated in [11]. Alternative ways of grouping SQL statements are also possible, as described in [6].

6.2.2. Tagger Pull-up Transformations

We now describe the XQGM transformations used during tagger pull-up. These transformations work in a bottom-up fashion on the initial query graph (such as Figure 10). Each transformation matches a fragment of the query graph and converts it to a semantically equivalent fragment in which the SQL part is separated from the tagger part. Repeated applications of these transformations produce the final separated query graph (such as Figure 11).

Our system implements tagger pull-up transformations for the various patterns that can occur in a query graph. Due to space constraints, we only describe one such transformation, which pulls up tagging for nested XML structures. Figure 12 depicts this transformation. The left side of the figure shows the XQGM fragment before the transformation. As can be seen, tagged elements are created (box 1), grouped into a list (box 2), and joined with their parent (box 3). This pattern would match boxes 4, 5 and 11 in Figure 10 (note that a left outer join is the same as a right outer join with the inputs flipped).

The right side of the transformation in Figure 12 shows the result of tagger pull-up. The merge (box 5) and aggregate (box 6) operators are used to create and group the nested elements, respectively. A merge operator (box 7) is used to relate the list of elements to their parent. A merge is sufficient for this purpose (instead of the join used earlier) because an ordering condition on parent id is pushed down to the SQL queries. A tagger input operator (box 4) is used to gate the children that belong to a certain parent, as described earlier.

The result of applying this transformation to boxes 4, 5 and 11 in Figure 10 produces boxes 3, 4 and 8 in Figure 11. A repeated application of this transformation would transform boxes 8, 9 and 12 in Figure 10 to boxes 6, 7 and 8 in Figure 11 (the duplicate box 8's created are merged using other query graph simplification rules).

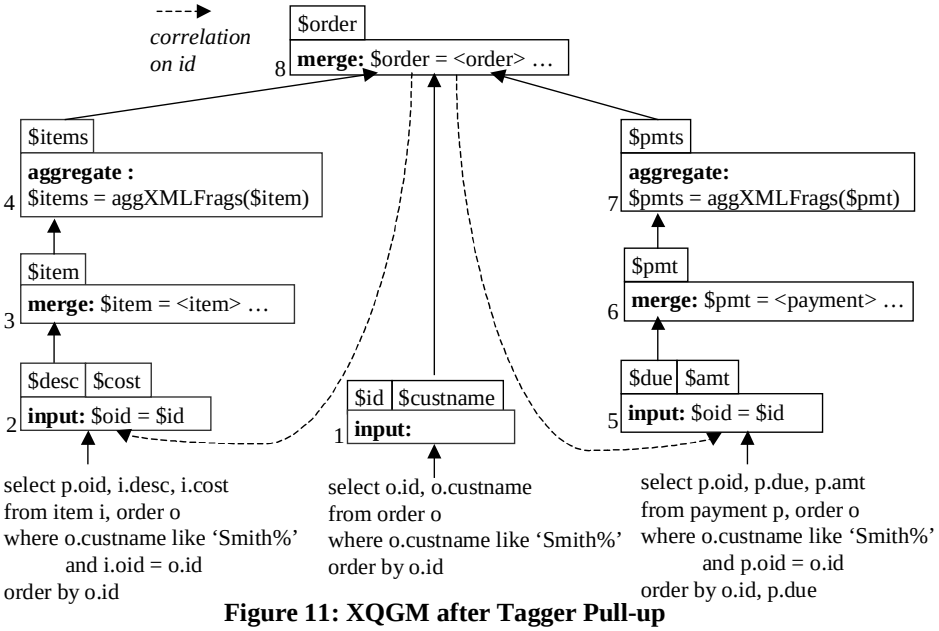

Figure 11: XQGM after Tagger Pull-up

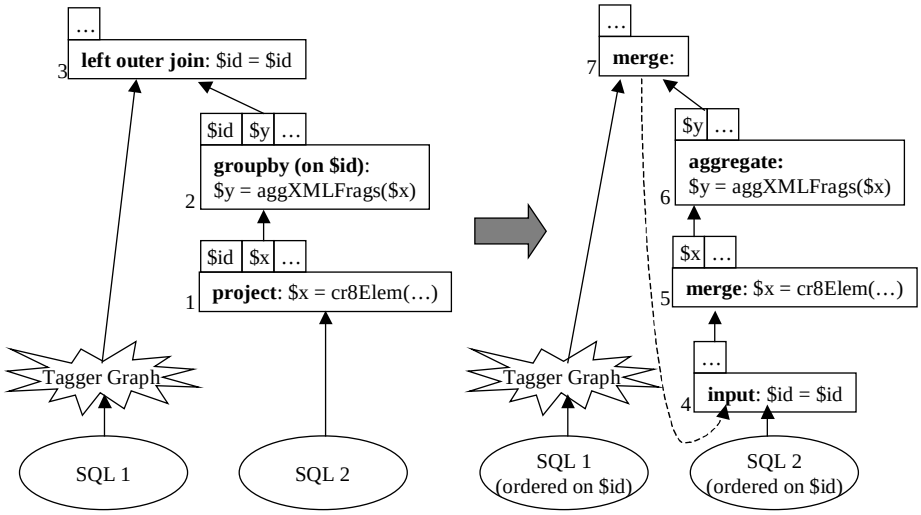

Figure 12: Tagger Pull-up Transformation

7. Implementation and Performance

We have implemented all of the techniques described above as part of the XPERANTO middleware system [3][12]. Our implementation is in Java and uses JDK 1.2. We use JDBC as the API to connect to a relational database system. As a result, our implementation works on top of most commercial database systems including DB2, Oracle and Microsoft SQL Server.

Based on our implementation, we have evaluated the performance of our proposed techniques. There are two factors that characterize the performance of queries in our context. The first is query compilation time, which consists of the time spent parsing the query, performing view composition, generating the SQL query and setting up the tagger run-time graph. The second is query execution time, which consists of the time spent executing the SQL query and tagging the SQL results to produce the output XML document. An earlier study [11] has evaluated query execution performance and has shown the superiority of the sorted outer union SQL plans that we generate. Hence, we only focus on query compilation time here.

Our experiments were performed using a 366 MHz Pentium II processor with 256MB of main memory running Windows NT 4.0. We used DB2 version 7.2 as our database system. We ran XPERANTO and the database system on the same machine to avoid unpredictable network delays. We considered queries that accessed up to 4 XML views. Each XML view nests 3 relational tables in a manner similar to Figure 3.

The compilation time for our experimental queries was always less than half a second. For instance, the compilation time for a query that accessed 12 relational tables through 4 XML views was about 450 milliseconds. It takes even less time to compile queries that access fewer views. As a result, there is a very small compile-time overhead associated with performing the optimizations proposed in this paper.

It is important to note that this small compile-time overhead is more than offset by the associated performance gains. This is due to two reasons. The first reason is that the view composition module eliminates the need to materialize intermediate XML fragments that do not appear in the final query result. As a result, only the relevant data is fetched from the relational engine. Thus, for typical queries that select only a small subset of data in an XML view, this results in many orders of magnitude improvement in performance. As a simple example, consider an XML view that publishes one million available items. If the user want details on only one of these items, it is clear that retrieving only the desired item will be orders of magnitude better than materializing all one million items and then selecting the desired one. This advantage is especially relevant when the underlying relational data changes often and cannot be easily cached in the middleware layer.

The other significant performance benefit is due to the computation pushdown module. By effectively harnessing the relational engine to process large parts of XML

queries, it eliminates the need for a full-fledged XML processor in the middleware layer – only a small, space-efficient tagger run-time mechanism is required. This is important because no existing native XML query processor has performance characteristics that are comparable to that of a parallel, scalable relational engine.

In the future, we plan to explore other performance enhancements such as pushing tagging inside the relational engine, as advocated in [11]. The next section presents more details regarding this.

8. Conclusion and Future Work

In this paper, we have focused on the problem of evaluating XML queries over XML views of relational data. In this context, we have described a general query-processing framework for processing arbitrarily complex nested XML queries. We have also described two techniques for efficiently evaluating XML queries. The first is a view composition mechanism that eliminates the construction of all intermediate XML fragments that do not appear in the final query result. The second is a computation pushdown mechanism that allows all data and memory intensive computation to be pushed down to the underlying relational engine as a SQL query.

However, as alluded to earlier in the paper, there are certain XML queries that cannot be directly pushed down to the relational engine. The first class of such queries are meta-data queries. These queries span relational meta-data (column and table names) and data (column values). While XQuery can naturally express such queries, SQL cannot. This is because SQL does not have certain higher-order operators [9]. Fortunately, it turns out that the desired higher-order operators can be provided in the middleware while still pushing most computation down to the relational engine (see [12] for more details).

The second class of queries that cannot be directly pushed down as SQL are those that perform user-defined operations on intermediate XML fragments. For example, consider a query that joins department and employee XML fragments using a user-defined XML predicate such as deptcontains(deptFrag, empFrag). It is important to note that the join predicate here involves XML fragments, and is not a predicate on basic data types such as integers (joins on basic data types can be handled using our computation push down mechanism). The reason that the join on XML fragments cannot be pushed down is because the relational engine does not know about XML fragment construction. A similar problem occurs when trying to order or group on XML fragments.

One solution is to perform these operations outside the relational engine, but this requires the duplication of sophisticated relational functionality, such as joins and sorts. Another solution, and the one we advocate, is to add primitives to construct XML document fragments inside the relational engine. In this way, all data and memory intensive processing can be done inside the relational engine. As shown in earlier work [11], the most efficient way to construct XML fragments inside the engine is to use the sorted outer union query plan. Integrating the computation pushdown technique with the relational engine so that these plans can be automatically generated is an area for future investigation.

Acknowledgements: We would like to thank Jeff Naughton for his valuable feedback throughout the course of this work. We would also like to acknowledge the contributions of past members of the XPERANTO project, including Rimon Barr, Mike Carey, Dana Florescu, Zack Ives, Ying Lu, and Subbu Subramanian. Finally, we would like to thank Gail Mitchell and the anonymous VLDB referees for their insightful comments.

9. References

[1] S. Banerjee, et. al., “Oracle8i – The XML Enabled Data Management System”, ICDE Conf., San Diego, March 2000, pp. 561-568.

[2] C. Baru, et. al., “XML-Based Information Mediation with MIX”, SIGMOD Conf. Demo, Philadelphia, June 1999.

[3] M. Carey, et. al., “XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents”, WebDB Workshop, Dallas, May 2000.

[4] J. Cheng, J. Xu, “XML and DB2”, ICDE Conf., San Diego, March 2000, pp. 569-573.

[5] V. Christophides, S. Cluet, J. Simeon, “On Wrapping Query Languages and Efficient XML Integration”, SIGMOD Conf., Dallas, May 2000, pp. 141-152.

[6] M. Fernandez, A. Morishima, D. Suciu, “Efficient Evaluation of XML Middleware Queries”, SIGMOD Conf., Santa Barbara, May 2001, pp. 103-114.

[7] M. Fernandez. W. Tan, D. Suciu, “SilkRoute: Trading Between Relations and XML”, World Wide Web Conf., Toronto, Canada, May 1999.

[8] L. Haas, J. Freytag, G. Lohman, H. Pirahesh, “Extensible Query Processing in Starburst”, SIGMOD Conf., Portland, May 1989, pp. 377-388.

[9] L. Lakshmanan, F. Sadri, I. Subramanian, “SchemaSQL – A Language for Interoperatability in Relational Multi-Database Systems”, VLDB Conf., Mumbai, India, Sep. 1996, pp. 239-250.

[10] Microsoft Corp. http://www.microsoft.com/XML.

[11] J. Shanmugasundaram, et. al., “Efficiently Publishing Relational Data as XML Documents”, VLDB Conf., Cairo, Egypt, Sep. 2000, pp. 65-76.

[12] J. Shanmugasundaram, et. al., “XPERANTO: Bridging Relational Technology and XML”, IBM Research Report, June 2001.

[13] P. Seshadri, H. Pirahesh, C. Leung, “Complex Query Decorrelation”, ICDE Conf., New Orleans, Louisiana, March 1996, pp. 450-458.

[14] World Wide Web Consortium, “The XML Query Algebra”, W3C Working Draft, 2001.

[15] World Wide Web Consortium, “XML Path Language (XPath) Version 1.0”, W3C Recommendation, 1999.

[16] World Wide Web Consortium, “XQuery: A Query Language for XML”, W3C Working Draft, 2001.