XTABLES: Bridging relational technology and XML

by J. E. Funderburk
G. Kiernan
J. Shanmugasundaram
E. Shekita
C. Wei

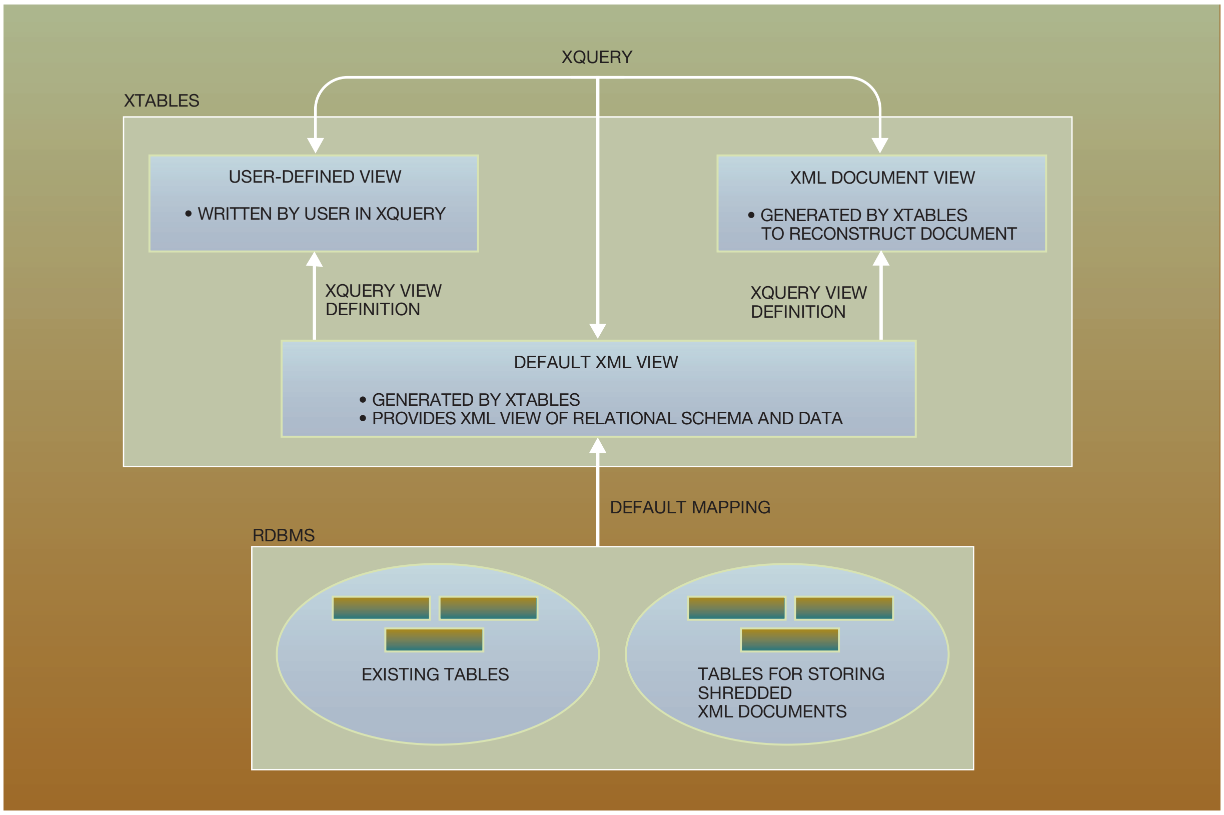
XML (Extensible Markup Language) has emerged as the standard data-exchange format for Internet-based business applications. These applications introduce a new set of data management requirements involving XML. However, for the foreseeable future, a significant amount of business data will continue to be stored in relational database systems. Thus, a bridge is needed to satisfy the requirements of these new XML-based applications while still using relational database technology. This paper describes the design and implementation of the XTABLES middleware system, which we believe achieves this goal. In particular, XTABLES provides a general framework to create XML views of relational data, query XML views, and store and query XML documents using a relational database system. Some of the novel features of the XTABLES architecture are that it (1) provides users with a single XML query language for creating and querying XML views of relational data, (2) executes queries efficiently by pushing most computation down to the relational database engine, (3) allows users to query seamlessly over relational data and meta-data, and (4) allows users to write queries that span XML documents and XML views of relational data.

Internet-based applications promise to dramatically reduce the cost of doing business by providing an automated and secure way to exchange data over the Internet. XML (Extensible Markup Language) 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 using relational database technology. This paper describes the architecture of the XTABLES middleware system, which we believe achieves this goal.

One of the features provided by XTABLES is the ability to create XML views of existing relational data. XTABLES does this by automatically mapping the schema and 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,[1] a general-purpose, declarative XML query language currently being standardized by the W3C (World Wide Web Consortium). XTABLES materializes XML views on demand, and does so efficiently by pushing down most computation to the underlying relational database engine.

Another feature provided by XTABLES is the ability to query XML views of relational data. This is important because users often need only a subset of a view's data. Moreover, users often need to synthe-

Figure 1 The XTABLES high-level architecture



size and extract data from multiple views. In XTABLES, queries are specified using XQuery, the same language used to specify XML views. XTABLES executes queries efficiently by performing XML view composition so that only the desired relational data items are materialized.

The final feature provided by XTABLES is the ability to store and query native XML documents. Users can query XML documents using the same query language that they use to create and query XML views of relational data. In addition, users can issue queries that span XML documents and XML views of relational data. As a result, users are provided with unified access to both relational data and XML documents, without having to deal with separate databases. XTABLES stores XML documents by automatically "shredding" them into rows in tables. Once again, queries over XML documents are efficiently executed by pushing most computation down to the relational database engine.

In summary, XTABLES provides a unified and general means to publish XML views of relational data and to store and query XML documents. Users always use the same declarative XML query language (XQuery) regardless of whether they are creating XML views of relational data, querying XML views, or querying XML documents. In addition, users can seamlessly query across XML views of relational data and XML documents. XTABLES runs as middleware on top of any relational database management system (RDBMS) and leverages the power of the underlying relational engine by pushing down most computation to it. The high-level architecture of the XTABLES system is depicted in Figure 1.

As discussed in the next section, there are other approaches that provide piecemeal solutions to the problems addressed by XTABLES. However, we believe that XTABLES is the first system to tie everything together in a unified framework. As a result of its general architecture, XTABLES also provides the following functionality not seen in other systems:

- *XML queries over relational data and meta-data.* The XTABLES default XML view captures both relational data and meta-data (schema) information. This allows users to write queries (and create views) that treat relational data and meta-data interchangeably. It provides users with a powerful "higher-order" query capability that is needed in certain applications.[2] Because SQL (Structured Query Language) identifiers are not always acceptable XML names, a mapping is defined from SQL identifiers to XML names. Our mapping algorithm follows the specifications defined in Reference 3.
- *Seamless queries across relational data and XML documents using a relational database system.* As mentioned earlier, XTABLES supports queries that span XML views of relational data and XML documents. This technique is general enough to work with all existing approaches for storing XML documents using a relational database system (for example, those described in References 4–6).

The main contributions of this paper are a high-level description of XTABLES's overall architecture, along with a more detailed description of the two features noted above. This paper complements References 7 and 8, which focus on individual components of XTABLES's query processor in more detail. The remainder of this paper is organized as follows. In the next section we discuss related work. In the following section, we outline our approach to creating and querying XML views. In the next two sections we describe meta-data querying and how XML documents are stored and queried in XTABLES. In the final section, we present our conclusions and outline avenues for future research.

Before proceeding, readers should note that this paper describes the research version of XTABLES, which previously went by the name of Xperanto.[9]

Comparison with alternative approaches

As noted earlier, there are other approaches that provide piecemeal solutions to the problems solved by XTABLES. In this section, we discuss some of those approaches and contrast them to XTABLES.

Querying XML documents using an RDBMS. There have been many approaches proposed for storing and querying XML documents using relational databases.[4–6] Each approach provides a different way to shred XML documents and store them as rows in tables. In addition, each approach provides its own query processor for translating XML queries to SQL queries. The XTABLES system does not provide yet another shredding algorithm and its associated query processor. Rather, XTABLES provides a general framework that allows the same query processor to be used irrespective of the shredding algorithm. XTABLES also supports seamless querying across XML documents and XML views of relational data.

Querying relational meta-data. XTABLES's ability to query relational meta-data is related to work on higher-order query languages such as SchemaSQL,[2,10–12] which is an extension of SQL. XTABLES's approach differs from this work in that an XML query language is used for querying meta-data. In addition, XTABLES uses the same query processor for both data and meta-data. However, there are similarities between the two approaches with respect to the final SQL queries generated.[11] Consequently, one promising direction for future research is to study the effectiveness of generating SchemaSQL instead of SQL in the XTABLES middleware layer.

Application programs. One approach for materializing an XML view of relational data is to write (or use a tool such as ODBC2XML** to generate) an application program to do the job. The main disadvantage of this approach is that *ad hoc* queries cannot be supported over XML views,[13] because application programs have to be written with a fixed set of parameters. Moreover, views cannot be defined on top of views in this approach.

XSL-T processors. XSL-T[14] processors can be used to create and query XML views of relational data. In order to do this, the desired relational data are first materialized in some interim XML format in which each row of a table is represented as a separate row element with the columns' values as subelements of a row, and then an XSL-T processor is used to transform the interim format to the desired XML view. Views over views and queries over views can be processed similarly. XTABLES differs from this approach in three respects. First, because 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 XTABLES, 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 XTABLES, intermediate views have to be materialized. Finally, query processing is less efficient in XSLT than in XTABLES. This is because XTABLES 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[15,16] is a system that allows users to create and query XML views of relational data. However, unlike XTABLES, it cannot store or query XML documents. 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.[17] Thus, unlike XTABLES, SilkRoute's users need to use different languages for creating and querying views. Further, its users need to explicitly straddle the relational and XML models by using the special-purpose RXL language. Also, because SilkRoute does not have the concept of a default XML view, it cannot support meta-data queries.

XML database systems. There have been special-purpose systems built for storing and querying XML documents.[18,19] XTABLES, on the other hand, uses relational databases for storing and querying XML documents. In addition, XTABLES allows users to publish existing relational data as XML documents, and also supports queries that seamlessly span relational data and XML documents. Although some XML database systems may also be capable of executing queries that span XML and relational data, these systems require the query to be decomposed into a composite execution plan—one part of the plan is run against the relational system, and the other against the XML engine—and the final result to be assembled in the XML database system. XTABLES, on the other hand, stores XML and relational data using the same database system, thereby tightly integrating the query processing over both forms of data.

XML integration systems. The part of XTABLES that provides the ability to create and query XML views of relational data can be used as a sophisticated wrapper for XML integration systems.[20–22] The role of integration systems is not to store data, but to integrate data coming from disparate sources. Since XML integration systems do not have any storage capability, they have to query native XML documents using an XML run-time engine in their integration layer. We, on the other hand, focus on storing XML data and are more aggressive in pushing down computation, because our focus is on relational systems with powerful query engines. This makes an order of magnitude difference in performance when constructing complex XML documents.[16,23]

Oracle. Oracle's XSQL[24] 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 supports the creation of XML views of relational data using object-relational technology.[25] 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, XTABLES provides a "pure XML" solution for querying XML documents.

SQL server. Microsoft's SQL Server[26] lets users create XML views of relational data using an XDR (XML data reduced) schema, which is a proprietary XML schema with relational annotations. Users can query an XDR-generated view, but in contrast to XTABLES, a different language (XPath[27]) is used for queries than is used for view definition. Moreover, queries are restricted to XPath expressions, which cannot support joins. Views on top of views are also not supported. Finally, there is no way to query XML documents using an XML query language.

DB2. IBM's Database 2* (DB2*)[28] allows users to create XML views of relational data using a proprietary mapping structure called a DAD (Document Access Definition), which is similar to IBM's XLE.[29] Users can invoke the DAD with parameters to materialize an XML view. However, there is no way to query the DAD or create XML views on top of the DAD. DB2 can also store XML documents in character BLOB (binary large object) columns. However, unlike XTABLES, only limited query support for XML documents is provided. Queries are restricted to XPath expressions. Finally, users cannot query across relational data and XML documents.

Creating and querying XML views of relational data

One of the key features of XTABLES is that users can create and query XML views of relational data using a standard XML query language, XQuery. We begin this section by describing how user-defined views are

Figure 2 A purchase order database and its default XML view

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	cost
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>
```

created in XTABLES. We then describe how queries over user-defined views are processed. The main purpose of this section is to help readers understand XTABLES'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 Reference 7.

The user's perspective. As a starting point, XTABLES automatically creates 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[15,26,28] in which a proprietary language is used to define the initial XML view of the underlying relational database. By using a general-purpose XML query language, XTABLES gains another important advantage; it allows arbitrarily complex views and queries to be expressed. This is in contrast to approaches[26,28] that provide limited querying capabilities without support for joins and recursion.

Figure 2 illustrates the default view for a simple purchase-order database. The database consists of three tables, one to track customer orders, a second to track items associated with an order, and a third to track the payments due for each order. Items and payments are related to orders by an order identifier (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[30] associated with the default view captures primary- and foreign-key relationships.

Continuing the example, suppose that 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 (for, let, where, return) 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[26] expression appearing in line 2 describes how to extract each "row" element from the order table: start at the root of the default view, navigate to each "order" element nested under it, and then navigate

Figure 3 XML purchase order

```
<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>
```

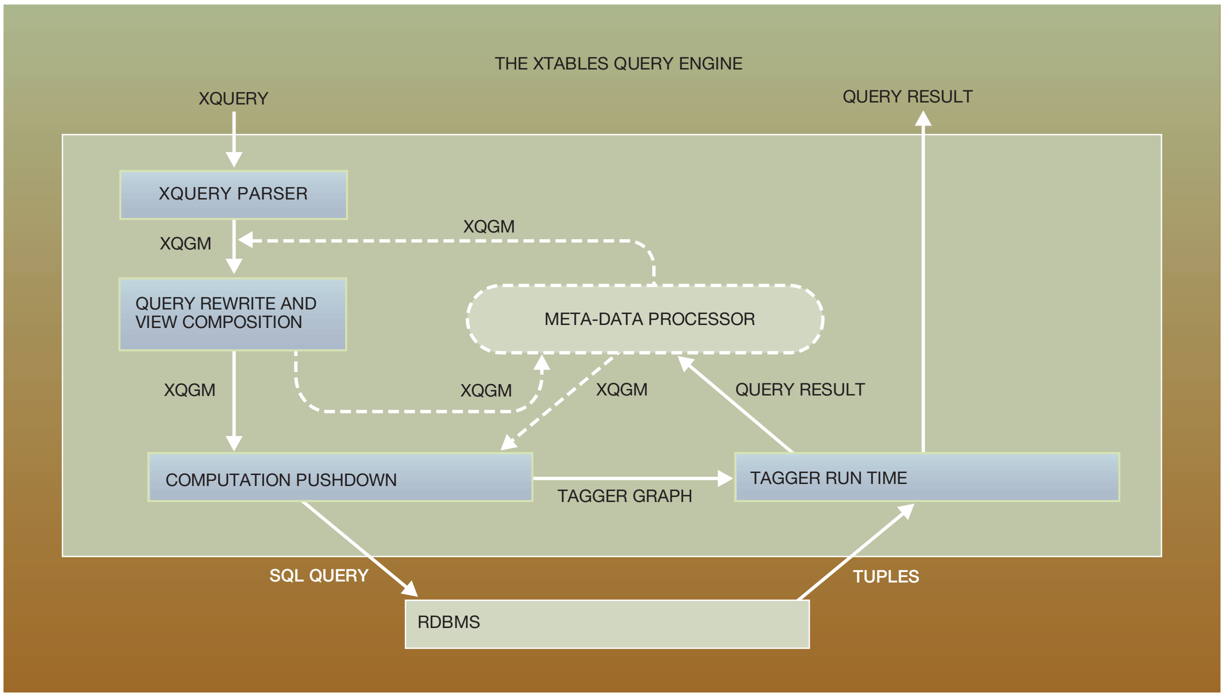
Figure 4 User-defined XML view

```
 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.  )
```

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.

Once the "orders" view has been created, queries can be issued against it. The following query extracts

Figure 5 Query processing architecture



a list of "item" elements from the "orders" view for the customer whose name begins with "Smith."

```
FOR $order IN view("orders")
LET $items = $order/items
WHERE $order/customer LIKE "Smith%"
RETURN $items
```

Query processing. In many ways, the heart of XTABLES is its query processor, which executes XQuery requests over XML views. One of the key features of XTABLES's query processing architecture is that it can perform view composition, so that only the desired relational data are materialized. This is in contrast to an approach such as XSLT,[13] where intermediate views have to be materialized. XTABLES's query processing architecture also allows it to harness the full power of the underlying relational database system by pushing most memory- and data-intensive computation down to the relational engine (see Reference 7 for a discussion on limitations of this approach for classes of queries having XML functions that cannot be pushed down to the relational engine, and possible relaxations of this restriction).

XTABLES's query processing architecture is shown in Figure 5. A query is initially parsed and converted from XQuery 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 most of 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 run-time module uses to construct the XQuery result in a single pass over the results of the SQL query. The result is then returned to the user. The steps in executing a query are depicted by the solid lines in Figure 5. Queries over relational meta-data follow a slightly different execution path, as illustrated by the dashed lines. A discussion of how meta-data queries are handled is deferred until the next section.

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 superset 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) as well as to project relational results.

The table and view operators in XQGM 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. A more complete presentation of XQGM can be found in Reference 7.

To provide a concrete example of how an XQuery is processed in XTABLES, we turn again to the query that 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 6. For clarity of exposition, variable names appearing in queries are used in XQGM representations of queries.

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 to 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, thus it can be easily adapted as the XQuery standard solidifies. Because the goal is to push down as much computation as possible to the relational engine, we purposely chose an intermediate representation that can be easily translated to SQL. Note that XQGM borrows from other intermediate representations for XML[21] as well as SQL.[31]

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, XTABLES 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.

Figure 6 The XQGM for a user query

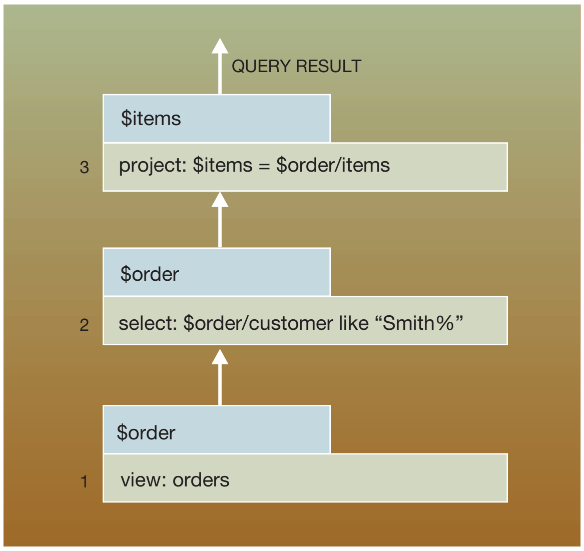
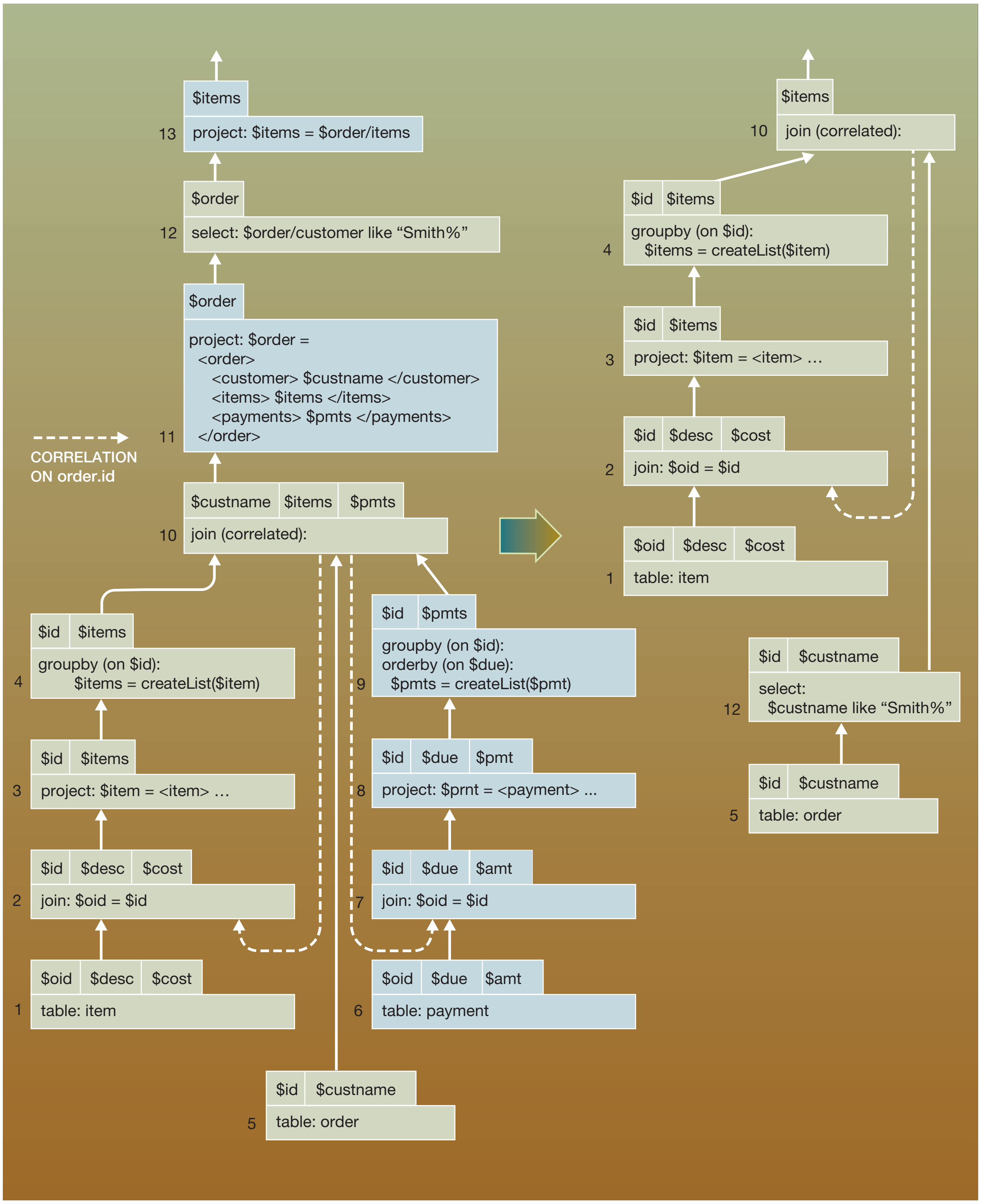

QUERY RESULT
$items
3
project: $items = $order/items
$order
2
select: $order/customer like "Smith%"
$order
1
view: orders

Table 1 XQGM operators

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
Group by	Applies aggregate functions and grouping
Order by	Sorts input based on column values
Union	Unions two or more inputs
Unnest	Applies superscalar functions to input
View	Represents a view
Function	Represents an XQuery function

Turning back to our example, Figure 7 illustrates the result of grafting the XQGM for the "orders" view in place of its reference. Box 1 in Figure 6 has now been replaced by boxes 1–11 in Figure 7. Focusing on the left side, 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) produced by the join is used to construct an "order" el-

Figure 7 XML view composition


$items
13 project: $items = $order/items
$order
12 select: $order/customer like "Smith%"
$order
11 project: $order = <order> <customer> $custname </customer> <items> $items </items> <payments> $pmts </payments> </order>
CORRELATION ON order.id
$custname $items $pmts
10 join (correlated):
$id $items
4 groupby (on $id): $items = createList($item)
$id $items
3 project: $item = <item> …
$id $desc $cost
2 join: $oid = $id
$oid $desc $cost
1 table: item
$id $pmts
9 groupby (on $id): orderby (on $due): $pmts = createList($pmt)
$id $due $pmt
8 project: $prnt = <payment> ...
$id $due $amt
7 join: $oid = $id
$oid $due $amt
6 table: payment
$id $custname
5 table: order
$items
10 join (correlated):
$id $items
4 groupby (on $id): $items = createList($item)
$id $items
3 project: $item = <item> …
$id $desc $cost
2 join: $oid = $id
$oid $desc $cost
1 table: item
$id $custname
12 select: $custname like "Smith%"
$id $custname
5 table: order

ement (box 11). The remainder of the XQGM (boxes 12–13) is the same as before.

The right side 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 other blue 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 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 most of 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 run-time 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 Reference 7. We show here the SQL query produced.

```
SELECT type, oid, custname, desc, cost
FROM (SELECT 0, order.id, order.custname, null, null
        FROM order
        WHERE order.custname like "Smith%"
        UNION ALL
        SELECT 1, order.id, null, item.desc, item.cost
        FROM order, item
        WHERE order.id = item.oid
) AS (type, oid, custname, desc, cost)
ORDER BY oid, type, due
```

XTABLES uses the "sorted outer union" technique, described in Reference 23, 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 identifier, type (0 = order, 1 = item), and due date. This allows the tagger run-time module to construct the output XML in a single pass over the SQL result.

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

Querying meta-data

One of the key features of XML is that it captures both data and meta-data. This feature is supported in XTABLES's default XML view, which captures both the data and meta-data (i.e., schema) of the underlying relational database. As a result, XTABLES users can treat relational data and meta-data interchangeably. In particular, they can query relational meta-data as though they were data. This kind of "higher-order" query capability is especially important in Internet-based business applications, where multiple XML views of the underlying relational database may be needed. Since the underlying relational schema may have been designed independently of these XML requirements, relational meta-data may have to be treated as data (and *vice versa*) in some XML views.[2,11]

Unfortunately, higher-order queries are not supported in standard SQL.[10] This creates a problem for XTABLES, because it must ultimately map XML queries to SQL. To solve this problem, XTABLES implements a higher-order function, *ExecXQuery*, in its query processor. This allows meta-data processing to be tightly integrated into XTABLES's query processor, allowing the same rewrite and computation pushdown techniques to be used for both data and meta-data queries.

In the remainder of this section, we first provide an example that illustrates the meta-data query problem. We then describe how meta-data queries are processed in XTABLES.

An illustrative example. Consider an industrial parts-supplier database that has the relational schema shown in Figure 8. The database contains tables that store information on three types of parts—resistor, capacitor, and voltmeter. These are stored in separate tables, because parts can have different columns depending on their type. For example, resistors have an "ohms" column, while capacitors have a "farads" column. In addition to the parts tables, there is also a category table that specifies the category of each part type. In our example, the resistor and capacitor part types appear under the electronic category,

Figure 8 An example relational schema

CATEGORY

catname	ptype
electronic	resistor
electronic	capacitor
test equip.	voltmeter

RESISTOR

sno	ohms	cost
1270	100	3.25
1763	500	4.75

CAPACITOR

sno	farads	cost
4671	5mf	2.23
4433	4mf	5.40

VOLTMETER

sno	farads	cost
3272	analog	22.00
3733	digital	85.75

Figure 9 The desired XML view

```
<parts>
  <part type="resistor"> <sno> 1270 </sno> <ohms> 100 </ohms> <cost> 3.25 </cost> </part>
  <part type="resistor"> <sno> 1763 </sno> <ohms> 500 </ohms> <cost> 4.75 </cost> </part>
  <part type="capacitor"> <sno> 4671 </sno> <farads> 5mf </farads> <cost> 2.23 </cost> </part>
  <part type="capacitor"> <sno> 4433 </sno> <farads> 4mf </farads> <cost> 5.40 </cost> </part>
</parts>
```

whereas the voltmeter part type appears under the test equipment category.

Let us now consider the case where the parts supplier wants to publish information about electronic parts, using the XML format shown in Figure 9. As shown, each "part" element has an attribute that specifies its type. Type-specific information is nested as subelements.

XTABLES users can easily achieve this task by creating a user-defined view over the default XML view of the parts database. Figure 10 shows the default XML view; the following query creates the user-defined view.

```
1. <parts>
2.  FOR $catrow IN view("default")/category/row
3.        $table IN view("default")"/*
4.        $part IN $table/row
5.  WHERE $catrow/catname = "electronic"
      AND $table/tagname( ) = $catrow/ptype
6.  RETURN <part type=$catrow/ptype> $part/* </part>
7. </parts>
```

The query works as follows. It first binds $catrow to all the "row" elements nested under the category "table" element in the default view (line 2). It then selects only those $catrow elements that correspond to the electronic category (line 5). This is done to determine the part types that belong to the electronic category. The query then binds $table to the "table" elements in the default XML view that correspond to electronic part types (resistor and capacitor). This is done by first getting all the table elements (line 3—note that "/*" gets all subelements), and then selecting those table elements that have the same tag name as one of the electronic part types (line 5). Once $table is bound to each electronic "table" element, all its "part" elements are determined (line 4) and the desired XML result is returned.

It is important to note that while XTABLES allows users to write such queries, SQL systems cannot directly support them. To see why this is the case, let us consider what a corresponding SQL query (to fetch all parts in the electronic category) would have to do. First, the SQL query would have to select from the category table to determine electronic part types. The resulting part types (resistor and capacitor) would be the *names of the tables* that belong to the electronic category. These dynamically determined tables would then have to be queried to get the actual parts. This presents a problem for SQL because table names in an SQL statement cannot be derived

Figure 10 The default XML view

```
<db>
  <category>
    <row> <catname> electronic </catname> <ptype> resistor </ptype> </row>
    <row> <catname> electronic </catname> <ptype> capacitor </ptype> </row>
    <row> <catname> test equip. </catname> <ptype> volt meter </ptype> </row>
  </category>
  <resistor>
     <row> <sno> 1270 </sno> <ohms> 100 </ohms> <cost> 3.25 </cost> </row>
     <row> <sno> 1763 </sno> <ohms> 500 </ohms> <cost> 4.75 </cost> </row>
  </resistor>
  <capacitor>
     <row> <sno> 4671 </sno> <farads> 5mf </farads> <cost> 2.23 </cost> </row>
     <row> <sno> 4433 </sno> <farads> 4mf </farads> <cost> 5.40 </cost> </row>
  </capacitor>
  ...
</db>
```

from data values, that is, they cannot be the result of subqueries. Thus, in our example, there is no way a single SQL query can determine the name of tables containing electronic parts *and then* query those tables.

This example illustrates how "higher order" queries can be supported in XTABLES. The default view effectively allows users to query seamlessly across data and meta-data using XQuery. This is in contrast to other approaches, which require application developers to either (1) use nonstandard query languages such as SchemaSQL,[10] or (2) write application programs that first issue a query to determine the desired table names, and then generate another query to retrieve the desired data items. The latter approach effectively moves part of query optimization into the hands of application developers, because a nondeclarative program must be written. It also prohibits queries over higher-order views. The next subsection describes how meta-data processing is tightly integrated into XTABLES's query processor, allowing the same rewrite and computation pushdown techniques to be used for both data and meta-data queries.

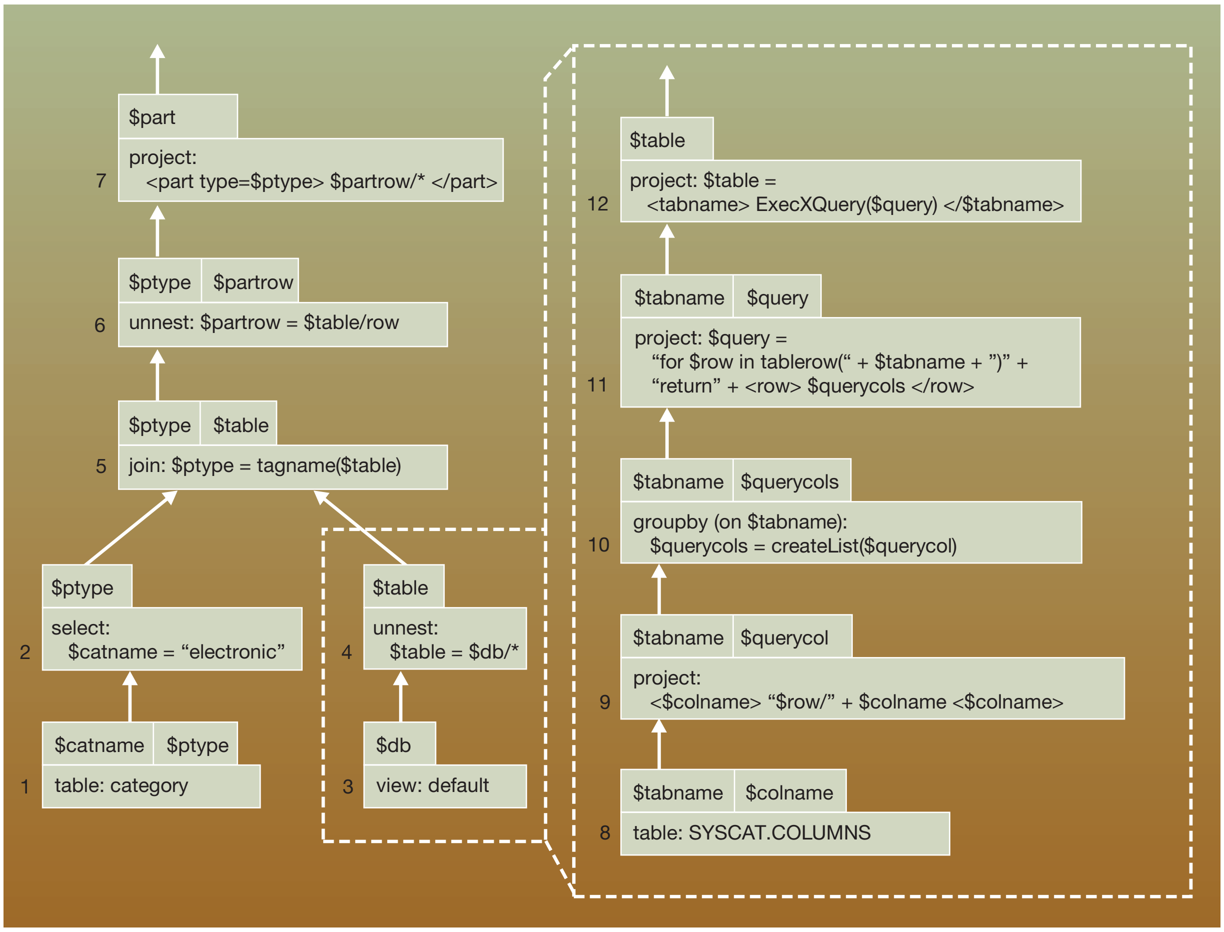
Meta-data query processing. To illustrate how meta-data query processing takes place in XTABLES, we turn back to the previous query. The XQGM for that query is provided in Figure 11. The figure gives the structure of the XQuery FLWR expression that is the main part of the example. As shown, electronic part types are first determined by performing a selection on the category table (boxes 1 and 2). These are joined with the tag names of "table" elements in the default view (box 5). The desired "part row" elements are then unnested from their "table" element (box 6), and used to produce the result (box 7).

The key feature to note is that the reference to the default view (box 3) cannot be directly replaced with references to the appropriate relational tables, as was done earlier (and also for the category table in box 1). This is because the names of the desired tables are data-dependent (a result of the join in box 5) and not known at query compile time.

The XTABLES query processor handles such queries by *representing the default XML view itself as a query* over the relational catalogs (which capture schema information). This approach has the following two advantages. First, it allows the same infrastructure to be used for processing both regular and meta-data queries. Second, as we shall soon see, it also allows a large part of the computation for meta-data queries to be pushed down to the relational engine. We now describe this technique in more detail, starting with how the default XML view is represented.

Representing the default XML view in XQGM. The XQGM corresponding to the query for the default XML view is shown in the right side of Figure 11. This corresponds to an expansion of boxes 3 and 4. At a high level, the operation corresponding to box 11 produces a query for each table in the relational da-

Figure 11 XQGM for the meta-data query and the default XML view


$part
7 project: <part type=$ptype> $partrow/* </part>
$ptype $partrow
6 unnest: $partrow = $table/row
$ptype $table
5 join: $ptype = tagname($table)
$ptype
2 select: $catname = "electronic"
$catname $ptype
1 table: category
$table
4 unnest: $table = $db/*
$db
3 view: default
$table
12 project: $table = <tabname> ExecXQuery($query) </$tabname>
$tabname $query
11 project: $query = "for $row in tablerow(" + $tabname + ")" + "return" + <row> $querycols </row>
$tabname $querycols
10 groupby (on $tabname): $querycols = createList($querycol)
$tabname $querycol
9 project: <$colname> "$row/" + $colname <$colname>
$tabname $colname
8 table: SYSCAT.COLUMNS

tabase. Each of these queries, *if executed*, materializes the "row" elements in the default view for the corresponding table. The top operation in the XQGM representation (box 12) invokes the higher-order function ExecXQuery to produce the result of executing each such query. In order to produce the default view, it then tags the result of the function invocation using the name of the corresponding table.

We now walk through an example to see how the default view is produced for a single table. First, all rows are retrieved from the relational catalog table SYSCAT.COLUMNS. Each row of this table represents a unique column in the relational database and includes the name of the column's table as well as the name of the column itself. Restricting our attention to the columns of the resistor table, this will produce the pairs (resistor, sno), (resistor, ohms), and (resistor, cost). The column names are then tagged and grouped on the table name (boxes 9, 10, 11) to produce the following query:

```
FOR $row IN tablerow(resistor)
RETURN <row><sno> $row/sno</sno> <ohms>
    $row/ohms </ohms><cost> $row/cost </cost>
    </row>
```

Here, tablerow is a new XQuery construct to bind directly over the rows of the specified relational table and not its default XML view. This query is then executed and tagged to produce the default view corresponding to the resistor table (box 12).

By representing the default view in XQGM form, we have essentially captured the relationship between meta-data (rows in SYSCAT.COLUMNS) and data (results of invoking ExecXQuery) within our query-processing framework (XQGM). Thus, by replacing references to the default view with the XQGM of the default view, the interaction between data and meta-data in the query can be explicitly captured and optimized.

Continuing our example, the meta-data-dependent reference to the default view can be replaced with the XQGM of the default view to produce the resulting XQGM, shown in Figure 12. This grafting also makes it possible to perform an important optimization. The meta-data-dependent join on the table name has been pushed below the invocation of the ExecXQuery function. (Notice that the join condition is independent of the result of the ExecXQuery function. Indeed, the $table variable used in the join condition in box 5 in Figure 11 is derived from the $tabname variable in box 12. Therefore, the join condition can simply be moved down through intermediate project boxes.) As in view composition, this is done after removing intermediate XML construction. By pushing down the join, we are essentially limiting the scope of the ExecXQuery function to exactly the tables the query actually refers to, that is, the resistor and capacitor tables. Although not shown in this example, other optimizations are also possible, such as pushing down predicates on column names. The challenge now is to implement the higher-order function ExecXQuery. We turn to this issue next.

Implementing ExecXQuery. To understand how ExecXQuery is implemented, we can draw an analogy between its invocation and the way that views are handled in SQL (and in XTABLES). Just as ExecXQuery logically produces the result of executing its input query, a reference to a view in SQL logically produces the result of executing the query used to define the view. Thus, invocations of ExecXQuery can be treated just as view references are treated in SQL. That is, just as references to an SQL view are replaced by the query used to define the view at query compilation time, invocations of ExecXQuery can also be replaced by (the XQGM of) its input query. The main advantage of grafting the XQGM of the query, rather than executing the query directly, is the same as in the case of SQL view composition—intermediate results do not have to be materialized. However, one issue has to be addressed before implementing ExecXQuery in the described manner. Unlike the query used to define an SQL view, the query passed as input to ExecXQuery is data-dependent. In our example, we need to determine the table names corresponding to electronic part types. Thus a query has to be first issued to determine the input query to ExecXQuery. This query is precisely the subquery represented as XQGM *below* the invocation of ExecXQuery (left side of Figure 12).

The key observation now is that the XQGM for the subquery is just like the XQGM for a regular query, that is, it does not have any higher-order operators. Therefore, *we can recursively invoke the XTABLES query processor to evaluate this query*. The query processor can thus perform optimizations, such as pushing computation down to the relational engine, for this subquery. The SQL generated during the evaluation of this subquery is

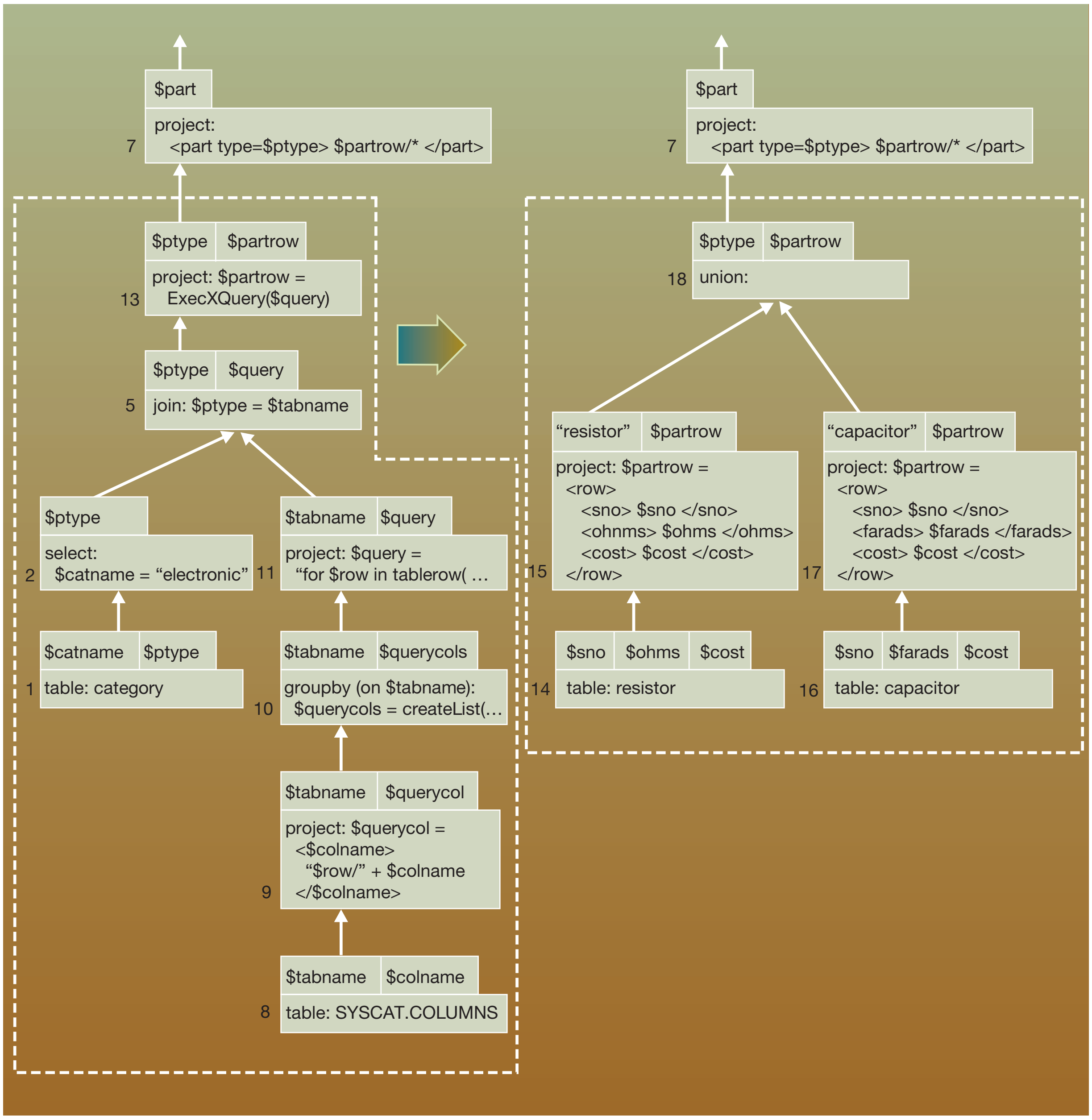
```
SELECT tabname, colname
FROM category c, SYSCAT.COLUMNS s
WHERE c.catname = “electronic” and c.ptype = s.tabname
ORDER BY tabname
```

As can be seen, it joins the category table with the relational catalog table to get the table and column information for all electronic part tables (resistor and capacitor in our example). The default view queries for these tables are then generated, parsed, and grafted in place of the invocation of ExecXQuery, as shown in the right side of Figure 12.

It should be noted that one of the key reasons XTABLES is able to reuse its query-processing infrastructure for meta-data queries is that relational systems allow catalog tables (SYSCAT.COLUMNS) to be queried just like other relational tables. This allows XTABLES to push down computation to the relational engine, even when the query is over relational catalog tables. Note, however, that just because relational database systems allow queries over their catalog tables, this does not imply that they support seamless querying across relational data and meta-data. Relational systems cannot use meta-data to construct and execute a new query “on the fly,” as is done in XTABLES using ExecXQuery.

Once ExecXQuery is eliminated, the resulting XQGM does not have any higher-order operators (right side of Figure 12). Hence, it can be processed like a regular query. The final SQL query generated by XTABLES for our example is

Figure 12 Query processing with ExecXQuery


$part
project: <part type=$ptype> $partrow/* </part>
7
$ptype $partrow
project: $partrow = ExecXQuery($query)
13
$ptype $query
join: $ptype = $tabname
5
$ptype
select: $catname = "electronic"
2
$catname $ptype
table: category
1
$tabname $query
project: $query = "for $row in tablerow(…
11
$tabname $querycols
groupby (on $tabname): $querycols = createList(…
10
$tabname $querycol
project: $querycol = <$colname> "$row/" + $colname </$colname>
9
$tabname $colname
table: SYSCAT.COLUMNS
8
$part
project: <part type=$ptype> $partrow/* </part>
7
$ptype $partrow
union:
18
"resistor" $partrow
project: $partrow = <row> <sno> $sno </sno> <ohmns> $ohms </ohms> <cost> $cost </cost> </row>
15
"capacitor" $partrow
project: $partrow = <row> <sno> $sno </sno> <farads> $farads </farads> <cost> $cost </cost> </row>
17
$sno $ohms $cost
table: resistor
14
$sno $farads $cost
table: capacitor
16

```
SELECT 0, "resistor", sno, ohms, cost, null, null, null
FROM resistor
UNION ALL
SELECT 1, "capacitor", null, null, null, sno, farads, cost
FROM capacitor
```

The query produces one row for each resistor and capacitor. These rows are unioned and a type field (first column) is used to distinguish resistors from capacitors. The part type information is also present (second column). As described in the previous section, this enables XTABLES's tagger run-time module to construct the output XML in a single pass over the SQL result.

Storing and querying XML documents

In the previous two sections, we described how XTABLES supports queries over XML views of relational data. In this section, we describe how XTABLES allows users to store and query XML documents using a relational database system. This problem has received much attention recently,[4–6] because it allows sophisticated relational storage and query technology to be used for XML documents.

Current approaches to storing and querying XML documents using a relational database system work as follows. The first step is *relational schema generation*, where relational tables are created for the purpose of storing XML documents. The next step is XML document *shredding*, where XML documents are "stored" by shredding them into rows of the tables that were created in the first step. The final step is XML query processing, where XML queries over the "stored" XML documents are converted into SQL queries over the created tables. The SQL query results are then tagged to produce the desired XML result.

The wealth of literature in this field[4–6] makes it clear that there are many possible approaches for relational schema generation. This is because the appropriate relational schema for a given application depends on many factors, such as the nature of data, the query workload, and availability of XML schemas. Currently, each relational schema generation technique has its own query processor for translating XML queries into SQL queries.[4–6] This is because there is no existing query processor that can provide a general query capability for all relational schema generation techniques.

The goal of the XTABLES system is not to provide yet another way to store and query XML documents using a relational database system. Rather, our goal is to provide a *single* query processor that can be used with *all* relational schema generation techniques. That would greatly simplify the task of relational schema generation by eliminating the need to write a special-purpose query processor for each new solution to the problem. In XTABLES, this goal is achieved using the same query processor as the one used for querying XML views of relational data. Since the same query processor is used to query XML views of relational data as well as XML documents, XTABLES can support queries spanning the two. As a result, XTABLES's architecture not only simplifies the task of relational schema generation, but also provides new functionality—the ability to seamlessly query over relational data and XML documents.

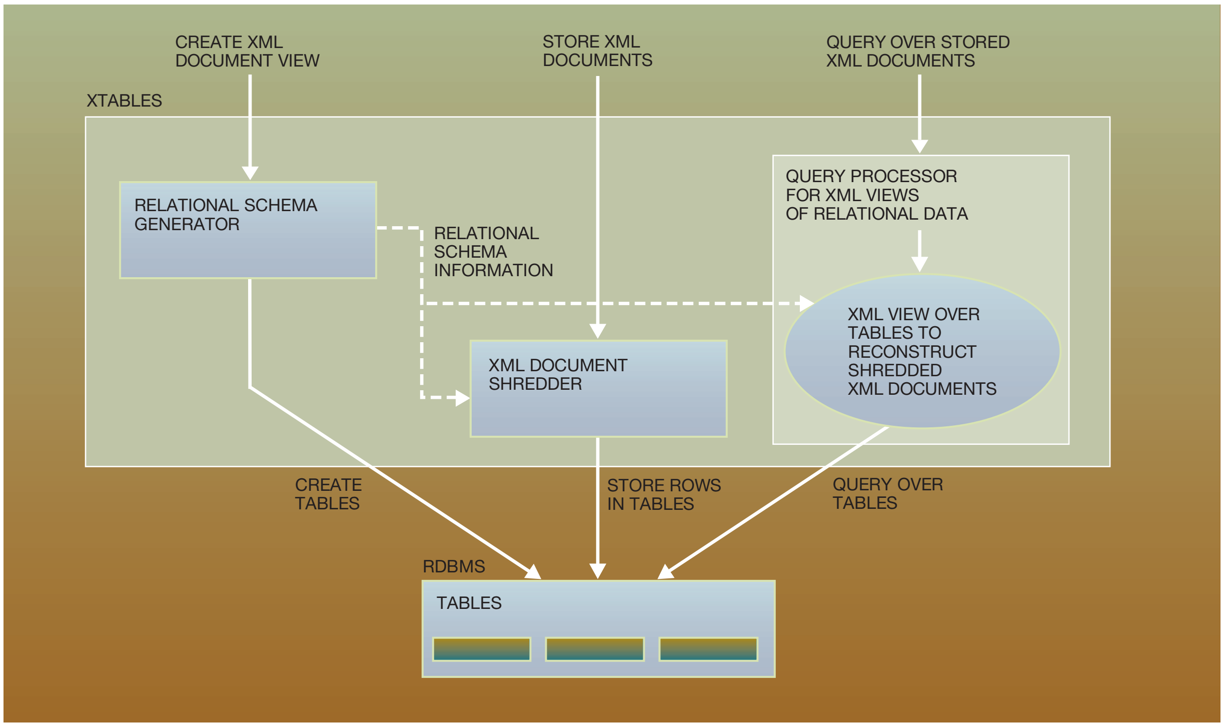
The rest of this section is organized as follows. We first describe XTABLES's architecture for storing and querying XML documents. We then walk through two illustrative examples to show how two very different approaches to relational schema generation can be handled using the XTABLES query-processing framework.

XML storage and querying architecture. In XTABLES, the first step in storing and querying XML documents is to create an XML document view. XML documents can then be stored in this view. Thereafter, XML document views can be treated just like regular XML views of relational data. For example, users can issue queries against XML document views. In addition, users can write queries that span XML document views and XML views of relational data.

XTABLES allows users to treat XML document views just like XML views of relational data because, internally, *XML documents are nothing but XML views of relational data*. Whenever a user creates an XML document view, XTABLES uses one of possibly many relational schema generators to automatically create relational tables for storing XML documents. XML documents "stored" in this view are then shredded and stored as rows in these tables. In addition, XTABLES generates a *reconstruction XML view* over the created relational tables, which (virtually) reconstructs the "stored" XML documents from the shredded rows. A reconstruction XML view is specified just like any other XML view of relational data—by using a query over the default XML view of the created tables.

A reconstruction XML view makes it possible to treat an XML document view as though it were an XML view of relational data. As a result, a query over an XML document view can be processed as a query over the reconstruction XML view. This in turn can be handled with the same query processor used for XML views of relational data. Further, since XTABLES's query processor can handle queries over multiple XML views, it becomes possible to process queries that span XML views of relational data and XML document views. This is shown pictorially in Figure 13.

Figure 13 Storing and querying XML documents in XTABLES


CREATE XML DOCUMENT VIEW
STORE XML DOCUMENTS
QUERY OVER STORED XML DOCUMENTS
XTABLES
RELATIONAL SCHEMA GENERATOR
RELATIONAL SCHEMA INFORMATION
QUERY PROCESSOR FOR XML VIEWS OF RELATIONAL DATA
XML VIEW OVER TABLES TO RECONSTRUCT SHREDDED XML DOCUMENTS
XML DOCUMENT SHREDDER
CREATE TABLES
STORE ROWS IN TABLES
QUERY OVER TABLES
RDBMS
TABLES

Recall that, in *relational schema generation*, tables are created for the purpose of storing XML documents. As mentioned earlier, the XTABLES architecture is general enough to support any technique for relational schema generation. This is possible because, for a given technique, only a program stub that does the following is required. When the stub is invoked (possibly with the schema of the XML documents to be stored), it

1. Generates the desired relational schema for storing XML documents
2. Produces an XML shredder object that can take in XML documents and shred them into rows in the tables of the generated relational schema
3. Creates a reconstruction XML view over the generated relational schema that indicates how shredded XML documents are to be (virtually) reconstructed

It is easy to see from Figure 13 how the above three components are sufficient to provide a general query capability over XML documents using any technique for relational schema generation. It is important to note that (1) and (2) have to be written, regardless of whether XTABLES is used. However, in XTABLES, it is sufficient to just generate a reconstruction XML view (3) rather than writing a full-blown XML query processor. The former is probably an order of magnitude easier to accomplish than the latter. As a result, XTABLES eliminates the need to build a new query processor for different relational schema generation techniques.

For the remainder of this section, we walk through examples to illustrate how the reconstruction XML view is relatively easy to generate for widely different relational schema generation techniques. In order to do this, we use two relational schema generation techniques published in the literature—one that uses XML schema information, and one that does not.

Case Study 1: Relational schema generation using an XML schema. In this subsection, we show how the relational schema generation technique proposed by Shanmugasundaram et al.[6] can be integrated with XTABLES. We briefly describe the proposed tech-

Figure 14 Creating an XML document view

```
Create XML Document View PurchaseOrderView using DTD {
  <!ELEMENT PurchaseOrder (ItemsBought, Payments)>
  <!ATTLIST  PurchaseOrder BuyerName CDATA #REQUIRED Date CDATA #REQUIRED>
  <!ELEMENT ItemsBought (Item)*>
  <!ELEMENT Item EMPTY>
  <!ATTLIST Item PartId CDATA #REQUIRED Cost CDATA #REQUIRED>
  <!ELEMENT Payments (Payment)*>
  <!ELEMENT Payment EMPTY>
  <!ATTLIST Payment CreditCard CDATA #REQUIRED ChargeAmt CDATA #REQUIRED>
}
```

nique and then show how an appropriate reconstruction XML view can be generated for it in XTABLES.

Relational schema generation and XML document shredding. The relational schema generation technique proposed in Reference 6 uses XML schema information[30] to create the appropriate relational tables. To illustrate how the technique works, consider an example of an XML document view definition in XTABLES, as shown in Figure 14. The body of the view specifies the DTD (Document Type Definition) of the XML documents to be stored. A description of the DTD specification is provided for readers unfamiliar with DTDs. The top-level element is called "PurchaseOrder." Each purchase order element has two subelements: "ItemsBought" and "Payments." Each purchase order element also has two attributes: "BuyerName" and "Date." Each "ItemsBought" element has zero or more "Item" elements, and each "Item" element has no subelements but two attributes. "Payments" elements are defined similarly.

Given the DTD information of the XML documents to be stored, the relational schema generation technique proposed in Reference 6 works as follows. First, a structure called the DTD graph, that mirrors the structure of the DTD, is created. The DTD graph for our example is shown in Figure 15. As can be seen, each node in the graph represents an XML element, an XML attribute, or an "operator." The "*" operator is used to identify "set" subelements, that is, those that can occur many times under a parent element.

Once the DTD graph is created, it is traversed to construct the desired relational schema. This is done by creating a relation for the root element of the DTD graph ("PurchaseOrder" in our example). All children of an element are represented in the same relation as the element, except children that are "*" nodes. Each such child corresponds to a "set" child, and because regular relations cannot capture set-valued attributes, the child of the "*" node is represented in a separate relation. Thus separate relations are created for the "Item" and "Payment" elements in our example.

The relational schema generated for our example DTD graph is shown in Figure 16. Note that all relations have an "id" field that serves as the primary key. In addition, all relations corresponding to nonroot elements ("Item," "Payment") also have a "ParentId" field that is a foreign key reference to its parent "PurchaseOrder." This is to link a child element to its parent element. Each relation corresponding to a nonroot element also has an order field that specifies the order in which the child elements appear under the parent element in the XML document.

The XML document shredder also uses the DTD graph to shred XML documents as rows in the generated tables. Figure 16 shows the rows obtained by shredding the XML document of Figure 17.

Reconstruction XML view generation. We now show how the technique described in the previous section can be integrated with XTABLES by providing an automatic mechanism to create reconstruction XML views. Recall that a reconstruction XML view is defined over the tables used to store shredded XML documents. It is used to reconstruct the original XML documents. This allows queries over XML documents to be treated as queries over the reconstruction XML view.

Figure 15 A DTD graph

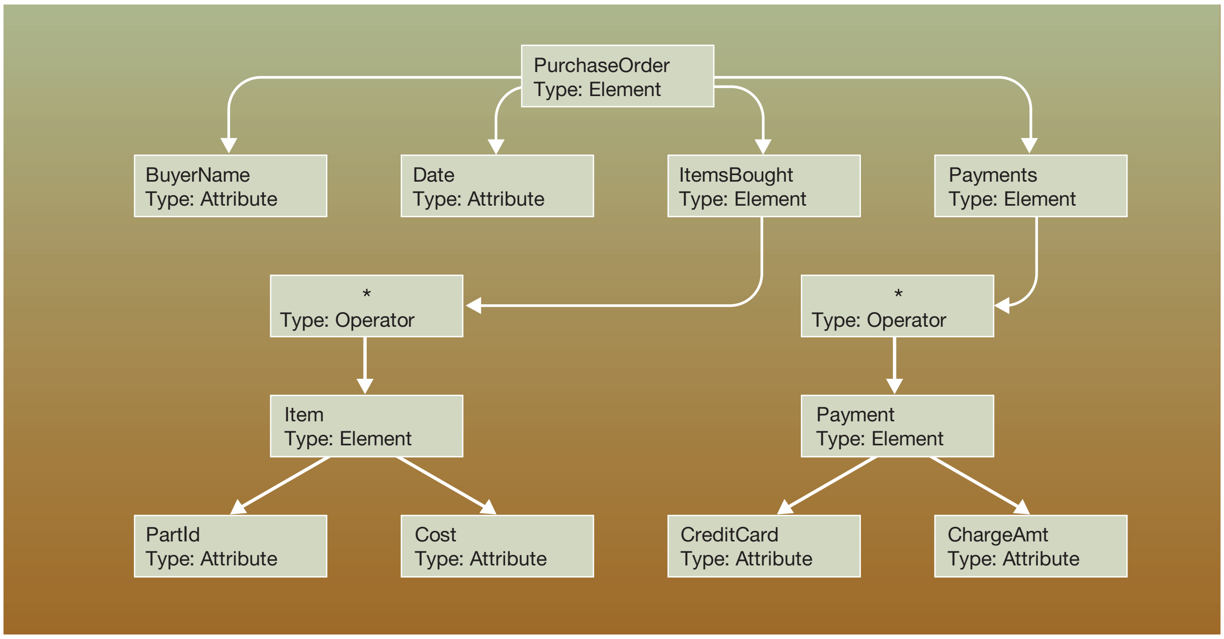
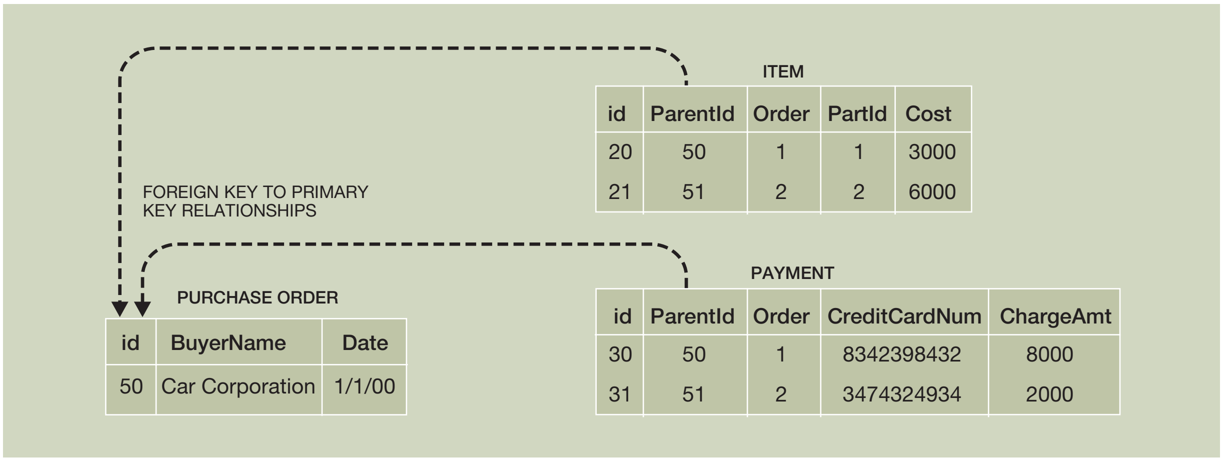


Figure 16 Generated relational schema



ITEM

id	ParentId	Order	PartId	Cost
20	50	1	1	3000
21	51	2	2	6000

PAYMENT

id	ParentId	Order	CreditCardNum	ChargeAmt
30	50	1	8342398432	8000
31	51	2	3474324934	2000

PURCHASE ORDER

id	BuyerName	Date
50	Car Corporation	1/1/00

Because a reconstruction XML view is an XML view over relational data, it can be expressed as a XQuery over the default view. The XQuery defining the reconstruction XML view for our example is shown in Figure 18. It reconstructs the XML document of Figure 17 from the rows in Figure 16. As shown, the query loops over all rows in the PurchaseOrder table to (re)construct the top-level "PurchaseOrder" XML elements. Nested queries are used to (re)construct "Item" and "Payment" subelements. Note that an "orderby" clause appears in the nested queries so that the subelements appear in the same order as they appeared in the original XML document.

Figure 17 Purchase order XML document

```
<PurchaseOrder BuyerName="Car Corporation" Date="1/1/00">
  <ItemsBought>
    <Item PartId="1" Cost= "3000"/>
    <Item PartId="2" Cost="6000"/>
  </ItemsBought>
  <Payments>
    <Payment CreditCardNum="8342398432" ChargeAmt="8000"/>
    <Payment CreditCardNum="3474324934" ChargeAmt="2000"/>
  </Payments>
</PurchaseOrder>
```

Figure 18 A reconstruction XML view

```
for $PurchaseOrder in view("default")/PurchaseOrder/row
return
  <PurchaseOrder BuyerName=$PurchaseOrder/BuyerName Date=$PurchaseOrder/Date>
     <ItemsBought>
       for $Item in view("default")/Item/row[ParentId = $PurchaseOrder/id]
       return  <Item PartId=$Item/PartId Quantity=$Item/Quantity Cost=$Item/Cost/>
       sortby($Item/Order)
     </ItemsBought>
     <Payments>
       for $Payment in view("default")/Payment/row[ParentId = $PurchaseOrder/id]
       return <Payment CreditCardNum=$Payment/CreditCardNum ChargeAmt=$Payment/ChargeAmt/>
       sortby($Payment/Order)
     </Payments>
  </PurchaseOrder>
```

Figure 19 presents the algorithm for creating a reconstruction XML view given an arbitrary DTD graph.[32] This algorithm works by recursively traversing the DTD graph, starting with the root node (PurchaseOrder in our example). If a separate relation has been created for a node in the relational schema, a new (sub)query is generated. In our example, separate queries are created for PurchaseOrder, Item, and Payment nodes. Nested queries are related to the parent query by joining on the ParentId field.

Note how the algorithm of Figure 19 eliminates the need for a special-purpose query processor for the relational schema generation technique proposed in Reference 6. In fact, using the generated reconstruction XML view, XTABLES provides a more powerful and efficient query capability than the query processor described there.

Case Study 2: Relational schema generation without an XML schema. We now show how a relational schema generation technique proposed by Florescu and Kossmann[5] can also be integrated with XTABLES. This technique, unlike the previous one, does not make use of XML schema information. As a result, the same reconstruction XML view can be used for any XML document with this technique. We first briefly describe the technique, and then show how reconstruction XML views can be generated over the generated relational schema.

Relational schema generation and XML document shredding. The basic idea behind this approach to relational schema generation is to view an XML document as a graph. The nodes of the graph are XML elements and attributes, and the edges represent containment relationships.[33] Each edge of this graph is

Figure 19 Algorithm to generate reconstruction XML view

```
Algorithm buildReconstructionQuery (DTDGraphNode node, String parentTableRowVariable) returns Query
  // First identify the type of the DTD Graph node
  if (node is of type Element) then
      // Check whether a separate table is created for this element node
      if (node is stored in separate table from parent) then
          // Create a new FLWR expression that creates a variable that ranges over the rows of the table
          QueryString = "For  $" + node.name + " in view(DefaultView)/" + node.name + "/row"
          // Join on parentId if this is not the root element
          if (not node.isRoot) then
              QueryString += "[parentId = " + parentTableRowVar + "/Id]"
          endif
          QueryString += "Return"
          currTableRowVariable = "$" + node.name
      else
          // Child is represented in the same table as the parent.
          currTableRowVariable = parentTableRowVariable;
       endif
       // Construct the element template by recursing on the attributes and sub-elements
       QueryString += "<" + node.name + " "
       for (all attributes A of node) do
           QueryString += buildReconstructionQuery(A, currTableRowVariable)
       endfor
       QueryString += ">"
       for (all sub-element and operator nodes E of node) do
           QueryString += buildReconstructionQuery(E, currTableRowVariable)
       endfor
       QueryString += "</" + node.name + ">"
       // If this element node is joined with its parent, then sort by the Order field
       if (node stored in separate table from.parent and not node.isRoot) then
           QueryString += "sortby (" + currTableRowVariable + "/Order)"
       endif
  else if (node is of type Attribute) then
       // Attributes are always in the same relation as the parent. So, just add attribute name = attribute value
       QueryString = node.name + " = " + parentTableRowVariable + "/" + node.name
  else // Node is of type Operator
       // Simply recurse on child
       QueryString = buildReconstructionQuery(node.child, parentTableRowVariable)
  endif
  // Return the query string built
  return QueryString
```

then stored in a relational table called the Edge table. Table 2 shows the Edge table populated with the edges of the XML document of Figure 17.

As shown, each edge is uniquely identified by the identifier fields of the source and destination nodes (the sid and did fields). Each edge also contains the name, value, and type information about its destination node. The order among sibling subelements is captured using the ordinal field. In our example, the edge pointing to the root XML element ("PurchaseOrder") is mapped to the first row. Its sid field is 0, which represents the identifier of the document root. The edges pointing to the BuyerName and Date attributes of the "PurchaseOrder" element are mapped to the second and third row, respectively. Note that these are related to the purchase order using the sid field. Similarly, the "ItemsBought" and "Payments" subelements of a "PurchaseOrder" element are represented by the fourth and fifth row, respectively. The ordinal field captures their relative order. The other edges of the document are stored similarly.

The reconstruction XML view. Figure 20 shows the query used to define the reconstruction XML view

Figure 20 The reconstruction XML view

```
 1.  function buildElement ($id integer, $name string, $value string) returns element {
 2.    <$name>
 3.      $value,
 4.      for $att in view("default")/Edge/row
 5.      where $att/sid = $id and $att/type = "Attribute"
 6.      return attribute($att/name, $att/value),
 7.      for $subelem in view("default")/Edge/row
 8.      where $subelem/sid = $id and $att/type = "Element"
 9.      return buildElement($subelem/did, $subelem/name, $subelem/value)
10.      sort by $subelem/ordinal
11.    </$name>
12.  }
13.  for $root in view("default")/Edge/row
14.  where $root/sid = 0
15.  return buildElement($root/did, $root/name, $root/value)
```

Table 2 The Edge table

sid	did	ordinal	name	value	type
1	0	0	PurchaseOrder	null	Element
2	1	null	BuyerName	Car Corp	Attribute
3	1	null	Date	1/1/00	Attribute
4	1	0	ItemsBought	null	Element
5	1	1	Payments	null	Element
6	4	0	Item	null	Element
. . .	. . .	. . .	. . .	. . .	. . .

for the relational schema generation approach just described. The query first determines the edge pointing to the root element and invokes a function called "buildElement" to construct the root element (lines 13–15). The buildElement function (lines 1–12) is recursive and builds up document fragments rooted at a given element. It first creates an element with the appropriate tags (line 2). It then produces the character values associated with an element (line 3). A nested subquery is then used to determine the edges pointing to the attributes of the element (lines 4–6), and the attributes are then created using the XQuery built-in function *attribute* (line 6). Finally, another nested subquery is used to determine the edges pointing to the subelement of the element (lines 7–8), and these are then created by recursively invoking the buildElement function (line 9). The subelements are then ordered by their ordinal position (line 10).

It should be clear that, given this reconstruction XML view, XTABLES can support queries over XML documents stored using this technique. Although not discussed in this paper, XTABLES is able to handle recursive queries, such as the one used in this reconstruction XML view.[7]

To summarize, this section describes XTABLES's approach to storing and querying XML documents using a relational database system. It also shows how any relational schema generation technique for storing XML documents can be used with XTABLES, and how its users can seamlessly query across XML views of relational data and XML documents.

Implementation and performance

We have implemented all of the techniques just described as part of the XTABLES middleware system.[9] Our implementation is in the Java** language and uses JDK** (Java Development Kit) 1.2. We use JDBC** (Java Database Connectivity) as the application programming interface 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's 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[23] has evaluated query execution performance and has shown the superiority of the sorted outer-union SQL plans that we generate. Hence, we focus only on query compilation time here.

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

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 four 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. First, the view composition module eliminates the need to materialize intermediate XML fragments that do not appear in the final query result. Thus, only the relevant data are fetched from the relational engine. 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 wants 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 Reference 23. The next section presents more details.

Limitations and possible extensions

We believe that the XTABLES system architecture can serve as the foundation for providing support for emerging XML query language features, such as updates.[34] We have already made some initial progress toward this goal by providing the ability to insert XML documents into a certain class of XML views. However, the development of a general theory of "updatable XML views" is an open research problem. Another interesting problem that arises in the context of XML query languages is support for user-defined XML functions and predicates. These raise new challenges because joins (or other memory-intensive operations) involving user-defined XML predicates cannot be pushed down to a relational engine. It is an open question as to whether a full-blown XML query engine is required in the middleware layer to efficiently answer such queries, or whether a relatively small run-time component that issues many SQL queries is sufficient. 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 pushdown 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,[22] 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.

Conclusion and future work

The emergence of a new wave of XML-based applications has imposed new requirements on data management systems. These requirements include publishing existing relational data as XML documents, and storing and querying native XML documents. This paper describes the design and implementation of the XTABLES middleware system that harnesses relational database technology to meet these requirements. XTABLES is unique in that it provides users with a unified, general, and declarative interface. Specifically, XTABLES exposes relational data as an XML view and also allows users to view native XML documents as XML views. Users can then query over 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 and can query seamlessly across XML views of relational data and XML documents. They can also query relational data and meta-data interchangeably. In addition to providing users with a powerful system that is simple to use, the declarative nature of user queries allows XTABLES to perform optimizations, such as view composition and pushing computation down to the underlying relational database system.

We believe that the XTABLES system architecture can serve as the foundation for pursuing various avenues of future research. One such area is providing support for emerging XML query language features, such as updates.[34,35] We have made some initial progress toward this goal by providing the ability to store or "insert" XML documents into a certain class of XML views. However, the development of a general theory of "updatable XML views" is an open research problem. Another interesting problem that arises in the context of XML query languages is providing support for information retrieval style queries. These are especially important for querying native XML documents.

There are also various optimization possibilities within the framework of the XTABLES architecture (in addition to those developed in this paper). For example, rather than reconstructing the result XML documents from the base relations every time, relevant fragments of the XML document can be cached. This optimization gains special significance in the context of storing native XML documents—if XML documents are stored as is, in addition to being shredded into tables, then the relational database can effectively be used as an index for XML documents. Other optimization opportunities include pushing down more functionality, such as tagging, to "XML-aware" relational databases and using SQL extensions such as SchemaSQL[10] as an alternative means to support meta-data querying. More generally, XTABLES can be used as a vehicle to determine what, and how much of, XML processing can effectively and efficiently be pushed down to a relational database system.

Acknowledgments

The authors would like to thank Rajasekar Krishnamurthy, Jeffrey Naughton, Efstratios Viglas, and Igor Tatarinov for their valuable contributions to the design and implementation of XTABLES.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Intelligent Systems Research, Sun Microsystems, Inc., Microsoft Corporation, or Intel Corporation.

Cited references and notes

1. World Wide Web Consortium, *XQuery: A Query Language for XML*, W3C Working Draft (February 2000). See http://www.w3c.org/TR/xquery.
2. R. Miller, "Using Schematically Heterogeneous Structures," *Proceedings, ACM SIGMOD International Conference on Management of Data*, Seattle, WA (June 2–4, 1998).
3. ISO-ANSI, *XML-Related Specifications (SQL/XML)*, (ISO-ANSI Working Draft) (June 2001). See http://www.sqlx.org/.
4. A. Deutsch, M. Fernandez, and D. Suciu, "Storing Semi-Structured Data with STORED," *Proceedings, ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA (June 1–3, 1999).

5. D. Florescu and D. Kossmann, “Storing and Querying XML Data Using an RDBMS,” *IEEE Data Engineering Bulletin* **22**, No. 3, 27–34 (1999).
6. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. Dewitt, and J. Naughton, “Relational Databases for Querying XML Documents: Limitations and Opportunities,” *Proceedings, 25th International Conference on Very Large Databases*, Edinburgh, Scotland (September 7–10, 1999).
7. J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk, “Querying XML Views of Relational Data,” *Proceedings, 27th International Conference on Very Large Databases*, Rome, Italy (September 11–14, 2001).
8. J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, E. Viglas, J. Naughton, and I. Tatarinov, “A General Technique for Querying XML Documents Using a Relational Database System,” *SIGMOD Record* **30**, No. 3 (September 2001).
9. M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanion, “XPERANTO: Publishing Object-Relational Data as XML,” *International Workshop on the Web and Databases (Informal Proceedings)*, Dallas, TX (May 18–20, 2000).
10. L. Lakshmanan, F. Sadri, and I. Subramanian, “SchemaSQL—A Language for Querying and Restructuring Multidatabase Systems,” *Proceedings, 22nd International Conference on Very Large Databases*, Bombay, India (September 3–6, 1996).
11. L. Lakshmanan, S. Sadri, and S. N. Subramanian, “On Efficiently Implementing SchemaSQL on a SQL Database System,” *Proceedings, 25th International Conference on Very Large Databases*, Edinburgh, Scotland (September 7–10, 1999).
12. L. Lakshmanan, F. Sadri, and I. Subramanian, “SchemaSQL—A Language for Interoperatability in Relational Multi-Database Systems,” *Proceedings, 22nd International Conference on Very Large Databases*, Bombay, India (September 7–10, 1996), pp. 239–250.
13. For the remainder of this paper, we use the terms “view” and “XML view” to mean “XML view of relational data” when the meaning is clear from the context.
14. World Wide Web Consortium, *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation (November 1999). See http://www.w3c.org/TR/xslt.html.
15. M. Fernandez, W. Tan, and D. Suciu, “SilkRoute: Trading Between Relations and XML,” *Proceedings, Eighth International World Wide Web Conference*, Toronto, Canada (May 11–14, 1999).
16. M. Fernandez, A. Morishima, and D. Suciu, “Efficient Evaluation of XML Middleware Queries,” *Proceedings, ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA (May 21–24, 2001).
17. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, “XML-QL: A Query Language for XML,” *Proceedings, Eighth International World Wide Web Conference*, Toronto, Canada (May 11–14, 1999).
18. R. Goldman, J. McHugh, and J. Widom, “From Semi-Structured Data to XML: Migrating the Lore Data Model and Query Language,” *Proceedings, ACM SIGMOD Workshop on the Web and Databases*, Philadelphia, PA (June 3–4, 1999).
19. J. Naughton et al., “The Niagara Internet Query System,” unpublished document, available at http://www.cs.wisc.edu/niagara/Publications.html.
20. C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu, “XML-Based Information Mediation with MIX,” *Proceedings, ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA (June 1–3, 1999).
21. V. Christophides, S. Cluet, and J. Simeon, “On Wrapping Query Languages and Efficient XML Integration,” *Proceedings, ACM SIGMOD International Conference on Management of Data*, Dallas, TX (May 16–18, 2000).
22. I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu, “XML and Relational: How to Live with Both,” Demonstration at the VLDB Conference, Cairo, Egypt (September 10–14, 2000).
23. J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsey, H. Pirahesh, and B. Reinwald, “Efficiently Publishing Relational Data as XML Documents,” *Proceedings, 26th International Conference on Very Large Databases*, Cairo, Egypt (September 10–14, 2000).
24. Oracle Corporation, http://technet.oracle.com/tech/xml.
25. S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy, “Oracle8i—The XML Enabled Data Management System,” *Proceedings, International Conference on Data Engineering*, San Diego, CA (February 26–March 3, 2000).
26. Microsoft Corporation, http://www.microsoft.com/xml.
27. World Wide Web Consortium, *XML Path Language (XPath) Version 1.0*, W3C Recommendation (November 1999). See http://www.w3c.org/TR/xpath.html.
28. J. M. Cheng and J. Xu, “XML and DB2,” *Proceedings, International Conference on Data Engineering*, San Diego, CA (February 26–March 3, 2000).
29. IBM Corporation, XML Lightweight Extractor, http://www.alphaworks.ibm.com/tech/xle.
30. World Wide Web Consortium, *XML Schema Part 0: Primer*, W3C Recommendation (May 2001). See http://www.w3.org/TR/xmlschema-0/.
31. H. Pirahesh, J. Hellerstein, and W. Hasan, “Extensible/Rule Based Query Rewrite Optimization in Starburst,” *Proceedings, ACM SIGMOD International Conference on Management of Data*, San Diego, CA (June 2–5, 1992).
32. The algorithm does not handle recursive DTD graphs. Although we have a general algorithm that handles recursion, we do not present it here, because the details are not particularly illuminating in the current context.
33. The Florescu and Kossmann paper (Reference 5) does not distinguish between attributes and subelements. However, since these are distinguished in the XML model, we treat them separately.
34. World Wide Web Consortium, *XML Query Requirements*, W3C Working Draft (August 2000). See http://www.w3c.org/TR/xmlquery-req.
35. I. Tatarinov, Z. Ives, A. Halevy, and D. Weld, “Updating XML,” *Proceedings, ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA (May 21–24, 2001).

Accepted for publication August 8, 2002.

John E. Funderburk *IBM Software Group, Silicon Valley Laboratory, 555 Bailey Avenue, San Jose, California 95141 (electronic mail: jfund@us.ibm.com).* Mr. Funderburk is a software developer at IBM's Silicon Valley Lab. He previously worked on DB2's XML Extender and is currently working on XTABLES.

Gerald Kiernan *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: kiernan@almaden.ibm.com).* Dr. Kiernan is a senior software engineer at IBM's Almaden Research Center. He previously worked on IBM's Object Broker, as well as the research version of XTABLES. He is currently doing research on privacy preserving databases.

Jayavel Shanmugasundaram *Cornell University, Department of Computer Sciences, Ithaca, New York 14853 (electronic mail: jai@cs.cornell.edu).* Dr. Shanmugasundaram is an assistant professor of computer science at Cornell University. He previously worked on the research version of XTABLES while he was a visiting scientist at IBM's Almaden Research Center. He is currently doing research on P2P indexing systems and query processing for unstructured data.

Eugene Shekita *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: shekita@almaden.ibm.com).* Mr. Shekita is a research staff manager at IBM's Almaden Research Center. He previously worked on DB2's query optimizer, as well as the research version of XTABLES. He is currently doing research on query processing.

Catalina Wei *IBM Software Group, Silicon Valley Laboratory, 555 Bailey Avenue, San Jose, California 95141 (electronic mail: fancy@us.ibm.com).* Ms. Wei is a senior software developer at IBM's Silicon Valley Lab. She previously worked on IBM's Object Broker and is currently working on XTABLES.