Efficient Evaluation of XML Middle-ware Queries

Mary Fernandez
AT&T Labs - Research
mff@research.att.com

Atsuyuki Morishima *
University of Tsukuba
mori@dblab.is.tsukuba.ac.jp

Dan Suciu †
University of Washington
suciu@cs.uwashington.edu

ABSTRACT

We address the problem of efficiently constructing materialized XML views of relational databases. In our setting, the XML view is specified by a query in the declarative query language of a middle-ware system, called SilkRoute. The middle-ware system evaluates a query by sending one or more SQL queries to the target relational database, integrating the resulting tuple streams, and adding the XML tags. We focus on how to best choose the SQL queries, without having control over the target RDBMS.

1. INTRODUCTION

XML is the universal data-exchange format between applications on the Web. Most existing data, however, is stored in non-XML database systems, so applications typically convert data into XML for exchange purposes. When received by a target application, XML data can be re-mapped into the application's data structures or target database system. Thus, XML often serves as a language for defining a *view* of non-XML data.

We are interested in the case when the source data is relational, and the exchange of XML data is between separate organizations or businesses on the Web. This scenario is common, because an important use of XML is in business-to-business (B2B) applications, and most business-critical data is stored in relational database systems (RDBMS). This scenario is also challenging, because the mapping from the relational model to XML is inherently complex and may be difficult to compute efficiently. Relational data is flat, normalized (3NF), and its schema is often proprietary. For example, relation and attribute names may refer to a company's internal organization, and this information should not be exposed in the exported XML data. In contrast, XML data is nested, unnormalized, and its schema (e.g., a DTD or XML Schema) is public. The mapping from the relational data to XML, therefore, usually requires nested queries, joins of multiple relations, and possibly integration of disparate databases.

In this work, we address the problem of evaluating efficiently an XML view in the context of SilkRoute [5], a relational to XML middle-ware system. In SilkRoute, a relational to XML view is specified in the declarative query language RXL. An RXL query has constructs for data extraction and for XML construction. We are interested in the special case of materializing large RXL views. In practice, large, materialized views may be atypical: often the XML view is kept virtual, and users' queries extract small fragments of the entire XML view. For example, SilkRoute supports composition of user-defined queries in XML-QL [4] and virtual RXL views and translates the composed queries into SQL. SilkRoute's query composition algorithm is described elsewhere [5]. Our goal is to support data-export or warehousing applications, which require a large XML view of the entire database. In this case, computing the XML view may be costly, and query optimization can yield dramatic improvements.

Shanmugasudaram et al. [9] evaluate experimentally a variety of approaches for publishing XML data in a relational query engine. In our scenario, the XML document defined by an RXL view typically exceeds the size of main memory, therefore, the *sorted, outer-union* approach [9] best suits our needs. This approach constructs one large, SQL query from the view query; reads the SQL query's resulting tuple stream; and then adds XML tags. The SQL query consists of several left-outer joins, which are combined in outer unions. The resulting tuples are sorted by the XML element in which they occur, so that the XML tagging algorithm can execute in constant space [9]. SilkRoute initially used a more naive approach, in which the view query was decomposed into multiple SQL queries that do not contain outer joins or outer unions. Each result is sorted to permit merging and tagging of the tuples in constant space. We call this the *fully partitioned* strategy.

This work makes two contributions. First, we show experimentally that neither of the above approaches is optimal. This is surprising for the sorted outer-union strategy, because only one SQL query is generated, and therefore has the greatest potential for optimization by the RDBMS. In experiments on a 100MB database, we found that the outer-union query was slower than the

*Research conducted as visitor at AT&T Labs.
†Research conducted as employee at AT&T Labs.

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

```
Supplier(*suppkey, name, addr, nationkey)
PartSupp(*partkey, suppkey, availqty)
Part(*partkey, name, mfgr, brand, size, retail)
Customer(*custkey, name, addr, nationkey, ph)
LineItem(*orderkey, partkey, suppkey, lno, qty, prc)
Orders(*orderkey, custkey, status, price, date)
Nation(*nationkey, name, regionkey)
Region(*regionkey, name)
```

Figure 1: Fragment of TPC-H Schema

queries produced by the fully-partitioned strategy. We found that the optimal strategy generates multiple SQL queries, but fewer than the fully partitioned strategy, therefore the optimal SQL queries may contain outer joins and outer unions. XML tagging still uses constant space, because it merges sorted tuple streams. The optimal strategy executes 2.5 to 5 times faster than the sorted outer-union and fully-partitioned strategies.

Given this finding, we want to devise an algorithm for decomposing an RXL view query into an optimal set of SQL queries. This problem is complicated by two issues. First, the RXL view query may be large, because it *constructs* an XML document and, therefore, is as complex as the output schema. Public DTD's have up to several hundreds elements and several thousand attributes, therefore any program or query generating XML documents for those DTDs must have a comparable complexity [7]. This rules out exhaustive-search strategies like the dynamic-programming algorithm of System R [8]. Second, our algorithm must function in a middle-ware system, and, therefore cannot rely on RDBMS-specific heuristics.

Our second contribution is a greedy optimization algorithm for the XML view-evaluation problem. The algorithm decomposes a RXL query into a set of SQL queries. The search algorithm is guided by *estimates* of query cost and data size provided by the RDBMS. We evaluated the algorithm on two views of the TPC/H database and found that, for both views, it generated the optimal strategies.

2. MOTIVATING EXAMPLE

We motivate the problem of efficient generation of XML views from databases with an example. We use the TPC Benchmark 'H' database [11], which contains information about parts, the suppliers of those parts, customers, and their part orders. Fig. 1 contains a fragment of the database's schema specified in datalog syntax. Key attributes are denoted by the '*' prefix. For example, the `Supplier` relation has four attributes and its key is the `suppkey` attribute.

We assume that information in this database needs to be exported in the format determined by the DTD in Fig. 2. This DTD specifies the XML format for the entire contents of the TPC database. Each `supplier` element includes its name, its `nation`, the geographical `region` of the nation, and a list of the supplier's `parts`. Each `part` element includes a part name and a list of orders pending for the part. Each `order` element includes an orderkey, the associated customer, and

```
<?xml encoding="US-ASCII"?>
<!ELEMENT suppliers (supplier*)>
<!ELEMENT supplier (name, nation, region, part*)>
<!ATTLIST supplier ID ID>
<!ELEMENT name     (#PCDATA)>
<!ELEMENT nation   (#PCDATA)>
<!ELEMENT region   (#PCDATA)>
<!ELEMENT part     (name, order*)>
<!ATTLIST part     ID ID>
<!ELEMENT order    (orderkey, customer, cnation)>
<!ATTLIST order    ID ID>
<!ELEMENT orderkey (#PCDATA)>
<!ELEMENT customer (#PCDATA)>
```

Figure 2: DTD of XML data about suppliers.

the customer's nation. The `name`, `nation`, `region`, and `customer` elements all contain strings.

To keep the example simple, we designed a DTD that follows naturally from the relational schema, but in practice, this may not be possible. DTDs for data exchange are created by agreement between partners and will not match each partners relational schema exactly. The DTD is also not unique: a different DTD might be specified by a public consortium of parts suppliers to provide access to order information for their customers. These requirements rule out automatic generation of the DTD or of the mapping between the relational schema and the DTD.

In SilkRoute, the mapping from the relational schema to the XML view is specified in RXL (Relational to XML transformation Language). RXL combines the extraction part of SQL (the `from` and `where` clauses), with the construction part of XML-QL (the `construct` clause). Fig. 3 contains the RXL query mapping the relational data to an XML output that is valid with respect to our DTD. As in SQL, the `from` clause declares *tuple variables* that iterate over tables. In this example, `$s` is a tuple variable that iterates over the `Supplier` table. The `where` clause contain conditions over these variables: for example `$s.nationkey = $n.nationkey` is a join condition. The `construct` clause specifies an XML fragment, which may contain expressions over the tuple variables.

RXL has three features that support creation of arbitrarily complex XML structures: nested queries, block structure, and Skolem functions. Nested queries occur inside `construct` clauses to construct sets of subelements. The block structure permits independent subqueries to construct different sets of elements, i.e., parallel blocks express union. For example, the outermost query in Fig. 3 has two sub-queries delimited by block boundaries `{...}`, each constructing a different set of elements. Skolem functions (not illustrated here) can be used to fuse objects constructed by different queries, which is useful in data integration.

To evaluate the RXL query computing the XML view, we must compute one or more SQL queries to extract and group the *data* for the XML view then add the XML *tags*. We focus here on generating the SQL queries. Each sub-query in the view definition corresponds to a SQL query, but they are correlated, and it is unclear

```
from Supplier $s
construct
  <supplier> <name>$s.name</name>
    { from Nation $n
      where $s.nationkey = $n.nationkey
      construct
        <nation>$n.name</nation>
        { from Region $r
          where $n.regionkey = $r.regionkey
          construct <region>$r.name</region> }  }
    { from PartSupp $ps, Part $p
      where $s.suppkey = $ps.suppkey,
            $ps.partkey = $p.partkey
      construct
        <part> <name>$p.name</name>
          { from LineItem $l, orders $o
            where $ps.partkey = $l.partkey,
                  $ps.suppkey = $l.suppkey,
                  $l.orderkey = $o.orderkey
            construct
              <order>
                <orderkey>$o.orderkey</orderkey>
                { from Customer $c
                  where $o.custkey = $c.custkey
                  construct
                    <customer>$c.name</customer>
                    { from Nation $n2
                      where $c.nationkey =
                            $n2.nationkey
                      construct
                      <cnation>$n2.name</cnation>
                    }
              }
              </order> }
        </part> }
</supplier>
```

Figure 3: Query 1 : RXL view of TPC-H.

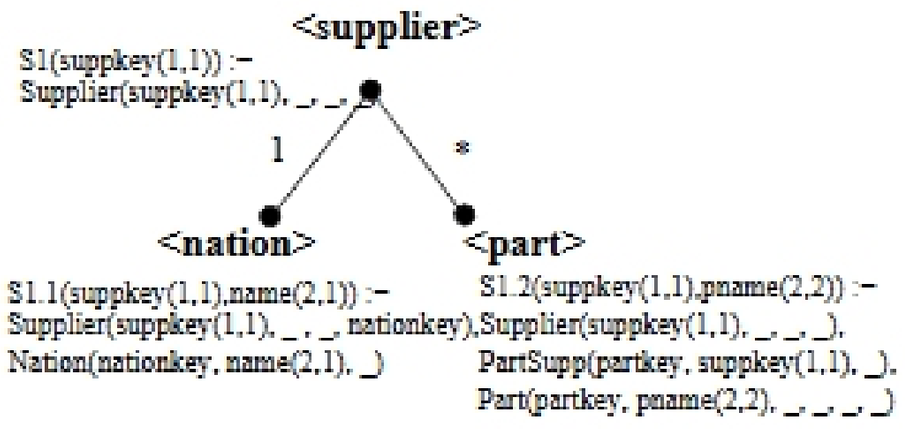

Figure 4: View tree for query fragment

how to put them together. To illustrate, we use the simpler RXL query contained in the boxes in Fig. 3.

The set of all possible choices are best visualized on the intermediate representation for RXL queries, which we call a *view tree*. Fig. 4 depicts the view tree for our simplified RXL query. Each node corresponds to an element in one of the `construct` clauses in the RXL query, and is annotated by a non-recursive datalog query that computes all instances of that node in the output XML. It is also possible to derive from the queries the multi-

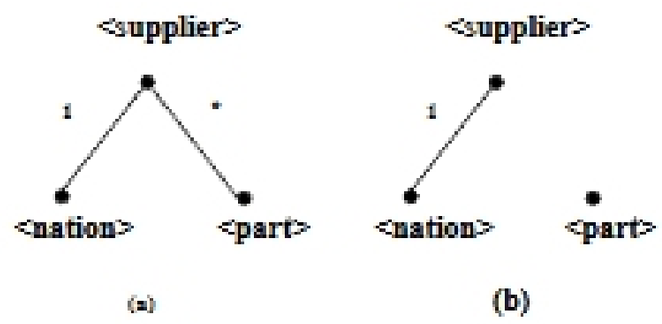

Figure 5: Execution plans for query fragment

plicities of the parent/child relationships, which are indicated by the labels 1 and *: the 1 between <supplier> and <nation> indicates that each <supplier> element in the output XML document will have exactly one child of type <nation>, and the * between <supplier> and <part> means that <supplier> may have zero or more many children of type <part>. Sec. 3.5 describes how to derive the multiplicities automatically.

The view tree makes it clear how to generate queries. A '1'-labeled edge requires an inner join, while a * requires a left outer join. Hence, the view tree corresponds to the following SQL query:

```
select s.suppkey, n.name, Q.partkey, Q.name
from Supplier s, Nation n
where s.nationkey = n.nationkey
left outer join
 (select ps.suppkey as suppkey, p.name as pname
  from PartSupp ps, Part p
  where ps.partkey = p.partkey
 ) as Q
on s.suppkey = Q.suppkey
order by s.suppkey
```

We need an outer join because there could be suppliers without parts, and they need to appear in the XML document. The `order-by` clause groups tuples from the same supplier together and allows the tagger to construct the <supplier> element using little memory.

We call the above query a "unified" translation, because it corresponds to the entire view tree and produces one relation. It is equivalent to the *sorted outer union* query in [9]. This is not the only choice. We can split the view tree into connected components, and generate a separate SQL query for each such component. Fig. 5 illustrates how to do this systematically: (a) corresponds to the query above, while (b), (c), and (d) are three alternative ways to partition the view tree into connected components. Each produces a set of SQL queries. For example, plan (b) results in the two SQL queries:

```
select s.suppkey, n.name
from   Supplier s, Nation n
where  s.nationkey = n.nationkey
```