Publishing Relational Data in XML: the SilkRoute Approach

Mary Fernández
AT&T Labs

Atsuyuki Morishima
Shibaura Institute of Technology

Dan Suciu
University of Washington

Wang-Chiew Tan
University of Pennsylvania

1 Introduction

For exchange on the Internet, relational data needs to be mapped to XML, a process that we call *XML publishing*. The mapping is complex, because the two data models differ significantly. Relational data is flat, normalized into many relations, and its schema is often proprietary. By contrast, XML data is nested, unnormalized, and its schema is public, usually created by agreement between members of a community, after lengthy negotiations. Publishing XML data involves joining tables, selecting and projecting the data that needs to be exported, mapping the relational table and attribute names into XML element and attribute names, creating XML hierarchies, and processing values in an application specific manner.

The XML data is a view over the relational database and, as such, can be *virtual* or *materialized*. In a virtual view, applications consuming XML data do so by applying XML queries to an XML virtual view. Some relational query processing is involved whenever the XML data is accessed. This is the most common scenario, because it guarantees data freshness and has the greatest potential to leverage the processing power of relational engines. In this scenario, one issue is the translation of an XML query of an XML view into SQL; this translation may be complex, because the XML view is itself complex. Another issue is that the automatically generated SQL queries often differ significantly from hand-written queries. In a *materialized* view, a large, unique XML document is created once from the relational data, requiring significant computation. Applications requesting the XML data always get the entire view, unless some external XML engine is available to extract portions of the XML data. This solution has trade-offs similar to those of a data warehouse: applications can access all the data without interfering with the relational engine, but the XML view needs to be refreshed periodically. In this scenario, an issue is the cost of computing the materialized view.

In this paper, we describe SilkRoute, a middle-ware system for publishing XML data from relational databases. In SilkRoute, an XML view is defined using a declarative query language called RXL (Relational to XML Transformation Language). RXL extracts data from the relational database and builds the resulting XML document by adding the appropriate element and attribute *tag* names, nesting elements and attributes, and processing values in an application-specific manner. The view can be either virtual or materialized. In virtual publishing, SilkRoute is in a query-processing mode: it accepts XML-QL queries over the XML view, translates them into SQL queries, sends these to the relational engine, tags the resulting tuple streams, and sends the XML result to the user. This mode of operation is described in detail in [6]. In materialized publishing, SilkRoute first optimizes the (usually complex) RXL query before sending it to the relational engine. There are several strategies

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

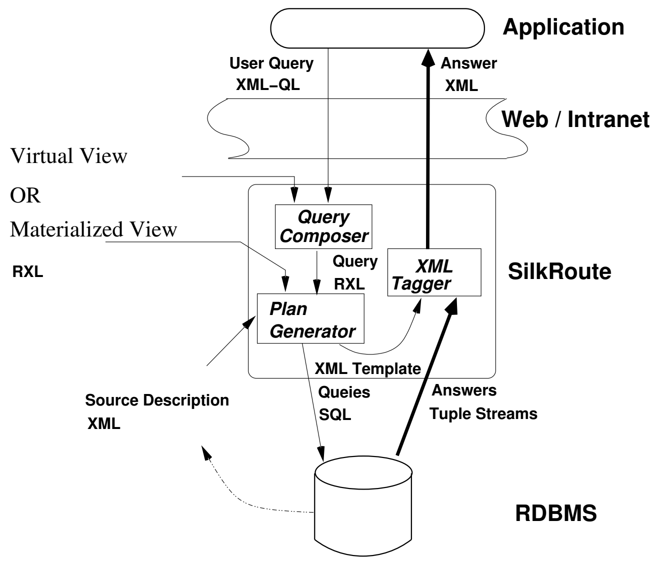

Figure 1: SilkRoute's Architecture.

to decompose a complex RXL query into a collection of SQL queries, and we found that the two simplest ones are significantly sub-optimal. SilkRoute deploys a greedy, heuristic optimization algorithm that decomposes the RXL query into multiple SQL queries. This mode of operation is described in detail in [5].

SilkRoute makes contributions to virtual and materialized XML publishing. In virtual publishing, the main contribution is a query composition algorithm that composes the view-definition query with the user's XML query. This is analogous to view expansion in relational databases, but is complicated by the complexity and hierarchical structure of the view and by the presence of wild-cards in the XML query. Here, we illustrate the algorithm on a simple example and refer the reader to [6] for details.

In materialized publishing, our main contribution is a heuristic algorithm for choosing an optimal execution plan for an XML view. The optimizer's choices range from constructing a single, large SQL query, to constructing many, relatively simple SQL queries both of which produce sorted tuple streams that are merged by an XML tagging module. We found empirically that the optimal plan lies somewhere between these two extremes. The optimizer faces two challenges. First, the size of the search space is, as usual, exponential in the size of the already large view definition. Second, the optimizer has no control over how the resulting SQL queries will be optimized by the relational engine. In SilkRoute, we developed a greedy, heuristic algorithm for finding an optimal query, that bases its decisions on query-cost estimates provided by the relational engine. Although the cost estimates can vary significantly from the real execution times, in our experiments, the greedy algorithm found near-optimal execution plans. We refer the reader to [5] for details.

2 SilkRoute's Architecture

SilkRoute is middle ware between a relational database (RDBMS) and an application accessing that data over the Web. Its architecture is depicted in Figure 1. The database administrator starts by writing an RXL query that defines the XML view of the database. This is called the *view query* and it is typically complex, because it transforms the relational data into a deeply nested XML view. The view can be either virtual or materialized. A

```
Clothing (pid, item, category, description,
          price, cost)
SalePrice(pid, price)
Problems (pid, code, comments)
```

(a)

```
<?xml encoding="US-ASCII"?>
<!ELEMENT supplier (company, product*)>
<!ELEMENT product  (name, category, descrip-
tion, retail,
                    sale?, report*)>
<!ATTLIST product  ID ID>
<!ELEMENT company  (#PCDATA)>
<!ELEMENT name     (#PCDATA)>
<!ELEMENT retail   (#PCDATA)>
<!ELEMENT sale     (#PCDATA)>
<!ELEMENT report   (#PCDATA)>
<!ATTLIST report code (size|defective|style) #REQUIRED>
```

(b)

Figure 2: Schema of supplier's relational database (a) and the DTD of the published XML view (b).

materialized view is fed directly into the Plan Generator, which generates a set of SQL queries and one XML template. A virtual view is first composed by the Query Composer with a user query (in XML-QL) resulting in another RXL query. A composed XML view is typically much smaller than the complete view definition, therefore plan generation is much simpler for composed views. The SQL queries are sent to the RDBMS server, which returns one sorted tuple stream per SQL query. The XML Tagger merges the tuple streams and produces the XML document, which is returned to the application.

SilkRoute mostly manipulates queries and delegates almost all data processing to the relational database engine. SilkRoute only touches data in the XML tagger, in which it merges multiple tuple streams and generates the result XML document in a single pass.

3 An Example

The Scenario We illustrate how to define an XML view in SilkRoute with a simple example from electronic commerce, in which *suppliers* provide product information to *resellers*. For their mutual benefit, suppliers and resellers have agreed to exchange data in a format that conforms to the DTD in Figure 2 (b). It includes the supplier's name and a list of available products. Each product element includes an item name, a category name, a brief description, a retail price, an optional sale price, and zero or more trouble reports. The contents of a retail or sale element is a currency value. A trouble report includes a code attribute, indicating the class of problem; the report's content is the customer's comments. Most importantly, this DTD is used by suppliers and resellers, and it is a public document.

Consider now a particular supplier whose business data is organized according to the relational schema depicted in Figure 2 (a). The `Clothing` table contains tuples with a product id (the table's key), an item name, category name, item description, price, and cost. The `SalePrice` table contains sale prices and has key field `pid`; the `Problem` table contains trouble codes of products and their reports. This is a third-normal form relational schema, designed for the supplier's particular business needs. The supplier's task is to convert its relational data into a valid XML view conforming to the DTD in Figure 2 (b) and make the XML view available to resellers. In this example, we assume the supplier exports a subset of its inventory, in particular, its stock of winter outer-wear that it wants to sell at a reduced price at the end of the winter season.

The View Definition Figure 3 contains the complete view query for our supplier example in Sec. 3, expressed in RXL. Skolem functions are used to control the creation and grouping of elements: for example, in `<product ID=Prod($c.pid)>` the function `Prod` is a Skolem function ensuring that a distinct `product`

```
 1. construct
 2. <supplier ID=Supp()>
 3.   <company ID=Comp()>"Acme Clothing"</company>
 4.    {
 5.     from  Clothing $c
 6.     where $c.category = "outerwear"
 7.     construct
 8.       <product ID=Prod($c.pid)>
 9.         <name  ID=Name($c.pid,$c.item)>$c.item</name>
10.         <category ID=Cat($c.pid,$c.category)>$c.category</category>
11.         <description ID=Desc($c.pid,$c.description)>$c.description</description>
12.         <retail ID=Retail($c.pid,$c.price)>$c.price</retail>
13.        { from SalePrice $s
14.          where $s.pid = $c.pid
15.          construct
16.          <sale ID=Sale($c.pid,$s.pid,$s.price)>$s.price</sale>
17.        }
18.        { from Problems $p
19.          where $p.pid = $c.pid
20.          construct
21.          <report code=$p.code ID=Prob($c.pid,$p.pid,$p.code,$p.comments)>
22.            $p.comments
23.          </report>
24.        }
25.        </product>
26.    }
27.  </supplier>
```

Figure 3: RXL view query.

element will be created for every distinct value of `$c.pid`. Lines 1, 2, and 27 create the root `<supplier>` element : notice that the Skolem term `Supp()` has no variables, meaning that a single `<supplier>` element is created. The outer-most clause constructs the top-level element `supplier` and its `company` child element. The first nested clause (lines 4–26) contains the query fragment described above, which constructs one `product` element for each "outerwear" item. Within this clause, the nested clause (lines 13–17) expresses a join between the `Clothing` and `SalePrice` tables and constructs a `sale` element with the product's sale price nested within the outer `product` element. The last nested clause (lines 18–24) expresses a join between the `Clothing` and `Problem` tables and constructs one `report` element containing the problem code and customer's comments; the `report` elements are also nested within the outer `product` element. Notice that the Skolem term of `product` guarantee that all `product` elements with the same identifier are grouped together. Usually Skolem terms are inferred automatically, but here we include them explicitly; they are used by the query-composition algorithm.

4 Virtual XML Publishing

In virtual publishing, the view definition query is not executed, but instead SilkRoute accepts XML-QL queries from applications that access the view. Users never see the relational data directly, but only through the XML view. Queries are formulated in XML-QL, a query language for XML [4].

The User Query The reseller can access that data by formulating queries over the XML view. Figure 4 shows an XML-QL query which retrieves all products with sale price less than half of retail price. The `where` clause consists of a pattern (lines 3-10) and a filter (line 11). A pattern's syntax is similar to that of XML data, but also may contain variables, whose names start with `$`. Filters are similar to RXL (and SQL). The meaning of

```
construct
 <results> {
   where <supplier>
           <company>$company</company>
           <product>
              <name>$name</name>
              <retail>$retail</retail>
              <sale>$sale</sale>
           </product>
         </supplier> in "http://acme.com/products.xml",
         $sale < 0.5 * $retail
   construct
         <result ID=Result($company)>
           <supplier>$company</supplier>
           <name>$name</name>
         </result>
 } </results>
```

Figure 4: XML-QL user query.

```
construct
  <results>
  { from  Clothing $c, SalePrice $s
    where $c.category = "outerwear",
          $c.pid = $s.pid,
          $s.price < 0.5 * $c.retail
    construct
      <result ID=Result("Acme Clothing")>
        <supplier ID=Supp("Acme Clothing")</supplier>
        <name ID=Name("Acme Clothing", $c.pid, $c.item)>
                  $c.item
        </name>
      </result>
  }
  </results>
```

Figure 5: Composed RXL query.

a query is as follows. First, all variables in the `where` clause are bound in all possible ways to the contents of elements in the XML document. For each such binding, if the filter condition is satisfied then the `construct` clause constructs an XML value. Grouping is expressed by Skolem terms in the `construct` clause. In this example, the construct clause produces one `result` element for each value of `$company`; each `result` element contains the supplier's name and a list of `name` elements containing the product names. It is important to notice that answer to the user query includes a small fraction of the relational database, i.e., only those products that are heavily discounted.

The Query Composer SilkRoute's query composer takes a user XML-QL query and composes it with the RXL view query resulting in a new RXL query. For lack of space, we only illustrate the query composition for our running example and refer the reader to [6] for details. Given the view query in Fig. 3 and the user query in Fig. 4, the composed query is given in Fig. 5. The composed query combines fragments of the view query and user query. Those fragments from the user query are highlighted. The composed query extracts data from the relational database in the same way as the view query. It also includes the user filter `$s.price < 0.5 * $c.retail` and structures the result as in the user query.

5 Materialized XML Publishing

In materialized XML publishing, we evaluate the view-definition query directly. In general, this is a relatively large query, which returns a large result, therefore choosing an optimal plan is critical. To illustrate alternatives, consider the view definition in Fig. 3. Fig. 6 (a) and (b) illustrate two strategies of computing this query in a relational engine. The first consists of three SQL queries returning three sorted tuple streams; the XML tagger merges the three sorted tuple streams and adds the XML tags. The second strategy is a single SQL query resulting in a single sorted tuple stream with tuples of two different types; tuples are coerced into a common type by padding them with null fields. In this case, the XML tagger reads a single tuple stream and just adds the XML tags.

In general, there are many ways in which an RXL query can be translated into a set of SQL queries, each generating a sorted tuple stream which the XML tagger merges and tags. The choices range from many, simple select-project-join SQL queries like in Fig. 3(a) to one large SQL query involving left outer joins and unions, like in Fig. 3(b). The latter may seem to be the best choice, since the relational query optimizer can always choose

```
select c.pid, c.item, c.category,
       c.description, c.price
from  Clothing c
where c.category = "outerwear"
order by c.pid

select c.pid, s.price
from  Clothing c, SalePrice s
where  c.category = "outerwear" and
       s.pid = c.pid
order by c.pid

select c.pid, p.code, p.comments
from  Clothing c, Problems p
where  c.category = "outerwear" and
       p.pid = c.pid
order by c.pid
```

(a)

```
select  c.pid, c.item, c.category,
        c.description, c.price,
        Q.price, Q.code, Q.comments
from
    ( select  c.pid, c.item, c.category,
              c.description, c.price
      from Clothing c
      where c.category = "outerwear")
    left outer join
    ((select s.pid as pid, s.price as price,
         null as code, null as comments
    from  SalePrice s)
    union all
   (select p.pid as pid, null as price,
      p.code as code, p.comments as comments
    from  Problems p)) as Q
on c.pid = Q.pid
order by c.pid
```

(b)

Figure 6: Two plans for computing the RXL view in Fig. 3.

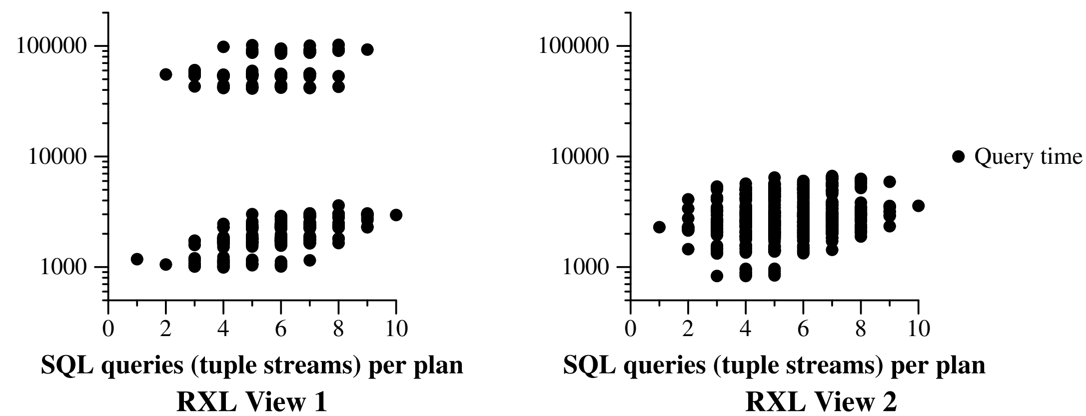

Figure 7: Query times (in msec) for all execution plans (1 MB database)

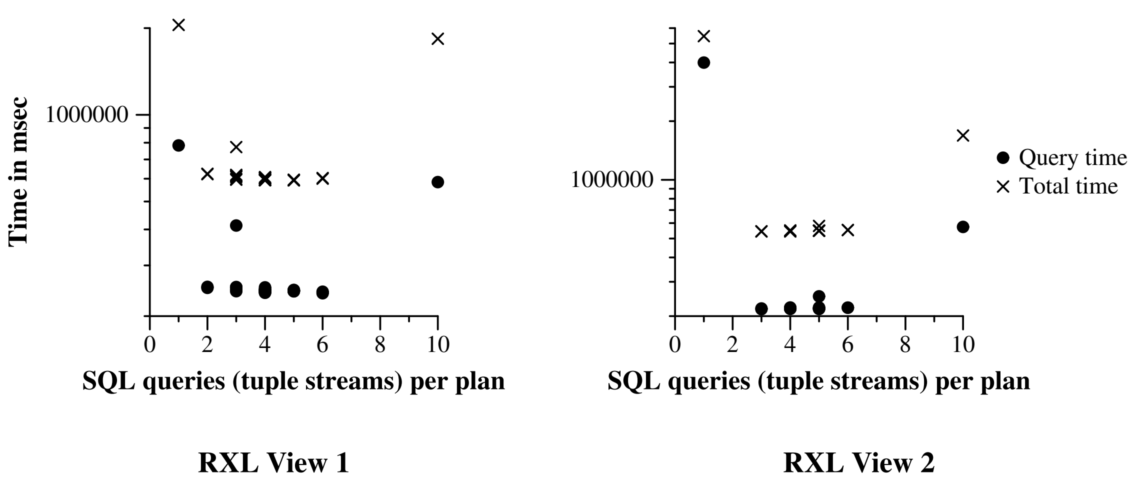

Figure 8: Query times for plans selected by greedy algorithm (100 MB database)

to implement it with multiple merge joins, thus mimicking all other strategies we may consider. However, we found at least one commercial RDBMS does not optimize effectively large SQL queries containing multiple outer joins. We considered two RXL view queries that create different XML views of the TPC/H benchmark. Each RXL view has $2^9 = 512$ translations into equivalent sets of SQL queries. We report their execution times on a 1 MB instance of the TCP-H database in Fig. 5 (queries that timed out after 5 minutes are not shown). From the graphs, we see that the two extreme queries (with one tuple stream, and with ten tuple streams respectively), were not optimal for either view. Instead, the optimum lies somewhere in the 4-6 tuple streams region.

In SilkRoute, we developed a greedy optimization algorithm that chooses an optimal set of SQL queries for a given RXL view definition. The algorithm bases its decisions on query cost estimations provided by the relational engine. The algorithm can be tuned to return more than one plan, allowing it to be integrated with additional optimization algorithms that optimize specific parameters, such as network traffic or server load. For a one MB instance of the TCP-H database, the greedy algorithm selected the top 32 fastest execution plans. For the 100 MB database, the greedy algorithm selected plans that were substantially faster than the two extreme plans. Fig. 5 reports the query time and total execution time (query + data transfer to SilkRoute) for the selected plans. Details of the greedy algorithm can be found in [5].

6 Discussion

Alternative Approaches We have described what we believe to be the most general approach for exporting relational data into XML. Other approaches are possible, and in some cases, may be more desirable.

Currently, the most widely used Web interfaces to relational databases are HTML forms with CGI scripts. User inputs are translated by a script into SQL queries, and their answers are rendered in HTML. The answers could be generated just as easily in XML. Forms interfaces are appropriate for casual users, but inappropriate for data exchange between applications, because they limit the application to only those queries that are predetermined by the form interface. Aggregate queries, for example, are rarely offered by form interfaces.

In another alternative, the data provider can either pre-compute the materialized view or compute it on demand whenever an application requests it. This is feasible when the XML view is small and the application needs to load the entire XML view in memory, e.g., using the DOM interface. However, pre-computed views are not dynamic, i.e., their data can become stale, and are not acceptable when data freshness is critical.

A third alternative is to use a native XML database engine, which can store XML data and process queries in some XML query language. XML engines will not replace relational databases, but a high-performance XML engine might be appropriate to use in data exchange. For example, one could materialize an XML view using SilkRoute and store the result in an XML engine that supports XQuery, thus avoiding the query composition cost done in SilkRoute. We don't expect, however, XML engines to match in performance commercial SQL engines anytime soon. In addition, this approach suffers from data staleness, and incurs a high space because it duplicate the entire data in XML.

Related Work Shanmugasundaram et al. [1, 8] describe an extension of IBM's DB2 relational engine to support XML publishing, with a focus on efficient computation of materialized views. This work considered a larger class of XML publishing techniques than does SilkRoute, including computing the XML view inside the relational engine. Overall, the in-engine methods perform better than the out-of-engine solutions.

Microsoft SQL Server 2000 allows users to define both virtual XML views and materialized XML views from a relational database [7]. Virtual views are defined in the XDR language. XDR is a specialized language for constructing XML views from a relational database, derived from an XML schema-definition language, and corresponds, in terms of expressive power, to a restricted fragment of RXL. Users can apply XPath [3] queries to the virtual view, which are translated by the system into relational queries. Alternatively, SQL Server 2000 offers a SQL extension called “FOR XML”, that allows users to define quite complex XML views, including some that are not expressible in RXL. However, the resulting views are always materialized.

Our virtual view approach is based on the XML-QL query language described in [4]. Currently, a standard query language for XML, XQuery [2], is being defined by the W3C. XQuery is more expressive and complex than XML-QL. An open research problem is whether query composition and decomposition is possible for the complete XQuery language or for only a subset of the language.

References

[1] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. subramanian. XPERANTO: publishing object-relational data as XML. In *Proceedings of WebDB*, Dallas, TX, May 2000.

[2] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: a query language for XML, 2001. available from the W3C, `http://www.w3.org/TR/query`.

[3] J. Clark. XML path language (XPath), 1999. available from the W3C, `http://www.w3.org/TR/xpath`.

[4] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eights International World Wide Web Conference (WWW8)*, pages 77–91, Toronto, 1999.

[5] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Santa Barbara, 2001.

[6] M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the WWW9*, pages 723–746, Amsterdam, 2000.

[7] M. Rys. Bringing the internet to your database: using SQLServer 2000 and XML to build loosely-coupled systems. In *Proceedings of the International Conference on Data Engineering*, pages 465–472, 2001.

[8] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as xml documents. In *Proceedings of VLDB*, pages 65–76, Cairo, Egipt, September 2000.