SilkRoute: Trading between Relations and XML

Mary Fernández   AT&T Labs – Research   `mff@research.att.com`
Wang-Chiew Tan   University of Pennsylvania   `wctan@saul.cis.upenn.edu`
Dan Suciu   AT&T Labs – Research   `suciu@research.att.com`

March 1, 2000

Abstract

Keywords. Data exchange, XML, Relational databases, XML Queries.

XML is the standard format for data exchange between inter-enterprise applications on the Internet. To facilitate data exchange, industry groups define public document type definitions (DTDs) that specify the format of the XML data to be exchanged between their applications. In this paper, we address the problem of automating the conversion of relational data into XML. We describe SilkRoute, a *general*, *dynamic*, and *efficient* tool for viewing and querying relational data in XML. SilkRoute is general, because it can express mappings of relational data into XML that conforms to arbitrary DTDs. We call these mappings *views*. Applications express the data they need as an XML-QL query over the view. SilkRoute is dynamic, because it only materializes the fragment of an XML view needed by an application, and it is efficient, because it fully exploits the underlying RDBMs query engine whenever data items in an XML view need to be materialized.

1 Introduction

XML can serve many purposes: as a more expressive mark-up language than HTML; as an object-serialization format for distributed object applications; or as a data exchange format. In this work, we focus on the role of XML in data exchange, in which XML documents are generated from persistent data then sent over a network to an application. Numerous industry groups, including health care and telecommunications, are working on document type definitions (DTDs) that specify the format of the XML data to be exchanged between their applications[1]. The aim is to use XML as a "lingua franca" for data exchange between inter-enterprise applications, making it possible for data to be exchanged regardless of the platform on which it is stored or the data model in which it is represented.

In this paper, we address the problem of exporting existing data into XML. Most data is stored in relational or object-relational database management systems (RDBMs) or in legacy formats. To realize the full potential of XML, we need tools that can automatically convert the vast stores of relational data into XML: we call the resulting XML document a *view*. We believe such tools must be *general, dynamic,* and *efficient*. Relational data is tabular, flat, normalized, and its schema is proprietary, which makes it unsuitable for direct exchange. In contrast, XML data is nested and unnormalized, and its DTD is public. Thus, the mapping from relational data to an XML view is often complex, and a conversion tool must be *general* enough to express complex mappings. Some commercial systems fail to be general, because they map each relational database schema into a fixed, canonical DTD. This approach is limited, because no public DTD will match exactly a proprietary relational schema. In addition, one may want to map one relational source into multiple XML documents, each of which conforms to a different DTD. Hence a second step is required to transform the data from its canonical XML form into its final XML form, e.g. using XSLT.

---

[1] Dozens of such applications can be found at `http://www.oasis-open.org/cover/`.

Our second requirement is that tools must be dynamic, i.e., only the fragment of the XML document needed by the application should be materialized. In database terminology, the XML view must be *virtual*. The application typically specifies in a query what data item(s) it needs from the XML document, and these items are typically a small fraction of the entire data. Some commercial products allow users to export relational data into XML by writing scripts. According to our definition, these tools are general but not dynamic, because the entire document is generated all at once.

Finally, to be efficient, such tools must exploit fully the underlying RDBMs query engine whenever data items in the XML view need to be materialized. Query processors for native XML data are still immature and do not have the performance of highly optimized RDBMs engines.

In this paper, we describe SilkRoute, a general, dynamic, and efficient tool for viewing and querying relational data in XML. SilkRoute is a particular instance of a mediator system, as defined by Geo Wiederhold [21]. In SilkRoute, data is exported into XML in two steps. First, an XML view of the relational database is defined using a declarative query language, called RXL (Relational to XML Transformation Language). The resulting XML view is virtual. Second, some other application formulates a query over the virtual view, extracting some piece of XML data. For this purpose, we use an existing XML query language, XML-QL. Only the result of that XML-QL query is materialized.

The core of SilkRoute is RXL, a powerful, declarative data-transformation language from flat relations to XML data. On the relational side, RXL has the full power of SQL queries and can express joins, selection conditions, aggregates, and nested queries. On the XML side, RXL has the full power of XML-QL, and can generate XML data with complex structure and with arbitrary levels of nesting. It can also specify arbitrary grouping criteria, using nested queries and Skolem functions. Typical RXL queries are long and complex, because they express general transformations from the relational store to the XML view. RXL has a block structure to help users organize, structure, and maintain large queries.

Once the virtual XML view is defined, SilkRoute accepts XML-QL user queries and *composes* them automatically with the RXL query. The result of the composition is another RXL query, which extracts only that fragment of the relational data that the user requested. Query composition is the most important technical contribution of this work, and our solution to this problem is general enough to be used in other systems requiring virtual XML views. In relational databases, composition is straightforward and not considered a problem; for example, Ramakrishnan [19] describes how to reformulate SQL queries over SQL virtual views as SQL queries over base relations. For XML, however, this problem is more complex, and, to the best of our knowledge, no solution has been published. We present a sound, complete, and conceptually simple algorithm that when given an RXL query and an XML-QL query, produces a new RXL query equivalent to their composition. We place some restrictions on aggregate functions in RXL queries, but they can be used freely in XML-QL queries.

Finally, when an RXL query is evaluated, most of the processing is done by the underlying relational engine. To do this, the RXL query is split into a collection of SQL queries, each of which produces a set of tuples. The SQL queries are sent to the RDBMS, and their flat, sorted results are merged in a single pass to construct the nested XML output.

In summary, this paper makes the following contributions:

- It describes a general framework for mapping relational databases to XML views to be used in data exchange.

- It describes a new query language, RXL, for mapping relational sources to XML views.

- It describes a sound and complete query composition algorithm that, when given an RXL query and an XML-QL query, generates a new RXL query equivalent to their composition.

- It describes a technique in which most of the work of an RXL query can be shipped to the underlying database engine.

To motivate this work, we present in the next section an example scenario from electronic commerce. In Section 2, we describe SilkRoute's architecture, its various components and the view-definition language,

```
<?xml encoding="US-ASCII"?>
<!ELEMENT supplier (company, product*)>
<!ELEMENT product (name, category, description, 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>
```

Figure 1: `supplier.dtd` : DTD of XML data exported by suppliers to resellers.

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

Figure 2: Schema of supplier's relational database. (* denotes key).

RXL. In Section 3, we give an informal discussion of our query composition algorithm by describing the pattern matching and query rewriting steps of the algorithm in detail. A brief discussion of aggregates appears in Section 3.4. Section 4 includes descriptions of related systems. A pseudo-code description of query composition can be found in Appendix A.

1.1 Motivating Example

We motivate 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 a particular DTD, depicted in Figure 1. 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. 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 schema is proprietary: for example, the supplier may not want to reveal the attribute `cost` in `Clothing`. The supplier's task is to convert its relational data into a valid XML view conforming to the DTD in Figure 1 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.

Once the XML views of suppliers' data are available, the reseller can access that data by formulating queries over the XML view. Some examples of such queries are:

- Retrieve products whose sale price is less than 50% of retail price.

- Count the number of "defective" reports for a product.

- Compute minimum and maximum cost of outer-wear stock.

As these queries suggest, the reseller is typically interested only in a small subset of the information provided by the suppliers. Readers familiar with SQL will recognize that these queries could be formulated as SQL
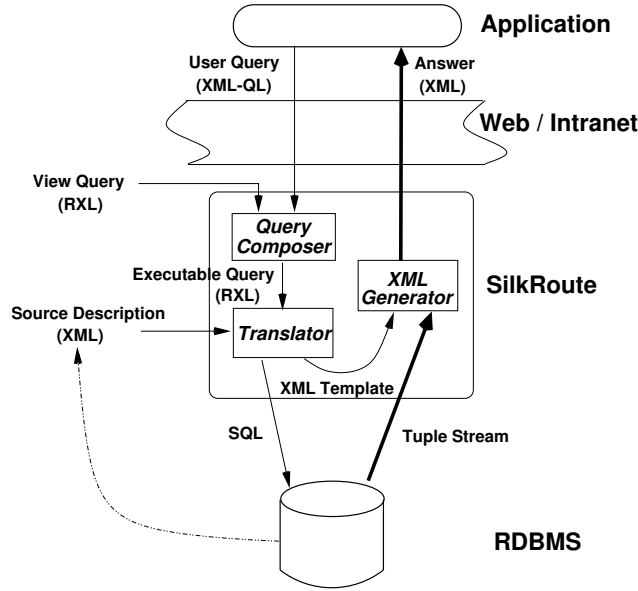
3

Figure 3: SilkRoute's Architecture.

queries over the supplier's relational database, but relational schemas differ from supplier to supplier and are not accessible by the reseller.

2 SilkRoute's Architecture

SilkRoute's architecture is depicted in Figure 3. It serves as middleware between a relational database (RDBMS) and an application accessing that data over the Web. The database administrator starts by writing an RXL query that defines the XML virtual 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 resulting view query is virtual, meaning that it is not evaluated, but kept in source code.

Typically, applications contact SilkRoute to request data. An application only "sees" the virtual XML view, not the underlying relational database. To access the data, it formulates an *user query* in XML-QL over the virtual view and sends it to SilkRoute. Together, the RXL view query and the XML-QL user query are passed to the *query composer*, the most complex module in SilkRoute. The composer computes the composition and produces a new RXL query, called the *executable query*. The answer to the executable query typically includes only a small fragment of the database, e.g., one data item, a small set of data items, or an aggregate value. Its result is an XML document, as specified by the user XML-QL query.

Once computed, the executable query is passed to the *translator*, which partitions it into a data-extraction part, i.e., one or more SQL queries, and an XML-construction part, i.e., an XML template. The translator also takes as input a description of the relational schema.

Until now, SilkRoute has manipulated only query source code, but no data. At this point, the SQL queries are sent to the RDBMS server, which returns one tuple stream per SQL query. The *XML generator* module merges these tuple streams and produces the XML document, which is then returned to the application.

This scenario is probably the most common use of SilkRoute, but minor changes to the information flow in Figure 3 permit other scenarios. For example, the data administrator may export the entire database as one, large XML document by materializing the view query. This can be done by passing the view query

4

directly to the translator. In another scenario, the result of query composition could be kept virtual for later composition with other user queries. This is useful, for example, when one wants to define a new XML view from an existing composed view. In the rest of this section, we describe the system's components in more detail.

2.1 The View Query: RXL

In this section, we describe RXL (Relational to XML transformation Language). RXL essentially combines the extraction part of SQL, i.e., a `from` and a `where` clause (possibly followed by `sort by` and/or `group by` clauses) with the construction part of XML-QL [9], i.e., the `construct` clause.

As a first example, consider this RXL query, which defines a fragment of an XML view:

```
from  Clothing $c
where $c.category = "outerwear"
construct  <product>
              <name>$c.item</name>
              <category>$c.category</category>
              <retail>$c.price</retail>
           </product>
```

Given a database like that in Figure 2, the query will produce an XML fragment like the following:

```
<product> <name>...</name> <category>...</category> <retail>...</retail> </product>
<product> <name>...</name> <category>...</category> <retail>...</retail> </product>
...
```

A root element is missing; we will explain later how to add one.

As in SQL, the `from` clause declares *variables* that iterate over tables. Variable names start with a $. In this example, $c is a tuple variable that iterates over the `Clothing` table. The `where` clause contains zero or more *filters* (boolean predicates) over *column expressions*. The column expression $c.item refers to the `item` attribute value of $c and in this case, requires that it equal the string "outerwear". The `construct` clause specifies the XML value, called an *XML template*, in terms of the bound column expressions.

RXL has three powerful features that make it possible to create arbitrarily complex XML structures: nested queries, Skolem functions, and block structure. An example of a *nested query* is:

```
construct <view> {
  from  Clothing $c
  construct <product>
              <name>$c.item</name>
              { from Problems $p
                where $p.pid = $c.cid
                construct <report>$p.comments</report>
              }
           </product>
} </view>
```

The outer query has no `from` or `where` clauses, only a `<construct>` clause for the root element `<view>`. The first sub-query builds one `<product>` element for each row in `Clothing`. Its inner sub-query creates zero or more `<report>` sub-elements, one for each report associated with that product. Readers familiar with SQL may recognize this as a left-outer join of `Clothing` with `Problems` followed by a `group by` on `Clothing`.

Skolem functions [14] allow us to control the way elements are grouped. Recall that in XML an attribute with type ID contains a value that uniquely identifies the element in the document, i.e., a *key*. In RXL, the distinguished attribute ID always has type ID, and its value is a Skolem term, which is used to control grouping and element creation. For example, in the following:

```
from  Clothing $c
construct <category ID=Cat($c.category) name=$c.category>
              <product>$c.item</product>
           </catebory>
```

`Cat` is a Skolem function and `Cat($c.category)` is a Skolem term whose meaning is that only one `<category>` element exists for every value of $c.category, and it includes all products in that category:

```
construct
  <view ID=View()>
    { from  Clothing $c
      construct <product ID=Prod($c.item)>
              <name ID=Name($c.item)>$c.item</name>
              <price ID=Price($c.item, $c.price)>$c.price</price>
            </product>   }
    { from  Clearance $d
      where $d.disc > 50
      construct <product ID=Prod($d.prodname)>
              <name ID=Name($d.prodname)>$d.prodname</name>
              <discount ID=Discount($d.prodname,$d.disc)>$d.disc</discount>
            </product>   }
  </view>
```

Figure 4: Multi-block RXL view query

```
<category><product>p1</product><product>p2<product></category>
<category><product>p3</product><product>p4</product></category>
...
```

Without the `ID` attribute and its Skolem term, the query would create one `<category>` element for each row in `Clothing`:

```
<category><product>p1</product></category>
<category><product>p2</product></category> ...
```

When Skolem terms are missing, RXL introduces them automatically. Since Skolem terms could be used to define arbitrary graphs, RXL enforces semantic constraints that guarantee a view always defines a tree, and therefore, a well-formed XML document. For example, the Skolem term of a sub-element must include all the variables of its the parent element.

Finally, the *block structure* allows RXL to construct parts of complex elements independently. The query in Figure 4 contains two blocks. The first block creates elements of the form:

```
<product><name>n</name><price>p</price></product>
```

for each product name in `Clothing`. The second block creates elements of the form:

```
<product><name>n</name><discount>d</discount></product>
```

for each product name in `Clearance`. (Here, we assume `Clearance(*prodname, disc)` is part of supplier's schema.) When the same product name occurs both in `Clothing` and `Clearance`, then the two elements will have the same ID key and are merged into:

```
<product><name>n</name><price>p</price><discount>d</discount></product>
```

Readers familiar with SQL will recognize this as an outer join.

Figure 5 contains the complete view query for our supplier example in Sec. 1.1. Lines 1, 2, and 27 create the root `<supplier>` element : notice that the Skolem term `Supp()` has no variables, meaning that one `<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 we include them explicitly, because they are relevant to query composition described in Sec. 3.

6

```
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</retail>
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 5: RXL view query (V)

2.2 The User Query: XML-QL

Applications do not access the relational data directly, but through the XML view. To do this, they write
user queries in XML-QL, a query language for XML [9]. XML-QL queries contain a `where` clause followed
by a `construct` clause. The `where` clause contains an arbitrary number of XML *patterns* and filters. The
`construct` clause is identical to that in RXL.

In our example, the reseller can retrieve all products with sale price less than half of retail price using
the XML-QL query in Figure 6. 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
$. Filters are similar to RXL (and SQL). The meaning of a query is as follows. First, all variables in the
`where` clause are bound in all possible ways to the contents of elements in the XML document. For each
such binding, the `construct` clause constructs an XML value. Grouping is expressed by Skolem terms in
the `construct` clause. In this example, the construct clause produces one `result` element for each value
of `$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.

2.3 The Query Composer

SilkRoute's query composer takes a user query and the RXL view query and generates a new RXL query,
which is equivalent to the user query evaluated on the materialized view.   In our example, the view query is in
Fig. 5, the user query in Fig. 6, and the composed query is in Fig. 7. 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. The details of composition are
subtle, and a complete description of the composition algorithm is given in Section 3.

We call the composed query *executable*, because it is typically translated into SQL queries and sent to the
relational database engine. Notice that the answer of the executable query is quite small — the same as that
of the user query. In general, it is more efficient to execute the composed query, instead of materializing the

7

```
1.  construct
2.   <results> {
3.     where <supplier>
4.             <company>$company</company>
5.             <product>
6.                 <name>$name</name>
7.                 <retail>$retail</retail>
8.                 <sale>$sale</sale>
9.             </product>
10.         </supplier> in "http://acme.com/products.xml",
11.         $sale < 0.5 * $retail
12.     construct
13.         <result ID=Result($company)>
14.             <supplier>$company</supplier>
15.             <name>$name</name>
16.         </result>
17.   } </results>
```

Figure 6: XML-QL user query (U).

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

Figure 7: Composed RXL query (C).

view query, because composed queries often contain constraints on scalar values that can be evaluated using indexes in the relational database. Such indices are of little or no use when evaluating a view query. For example, consider a user query that specifies the condition: `$s.price between 80 and 100`. This condition is propagated into the executable query, and then into the SQL query, and can be evaluated efficiently if an index exists on `price`. In contrast, an index on `price` is useless when materializing the view query directly.

2.4 Translator and XML Generator

The *translator* takes an RXL query and decomposes it into one or more SQL queries and an XML template. The SQL queries are executed by the relational engine, and their flat results (streams of tuples) are converted into XML by the *XML generator*.

The translator also takes a source description, which is an XML document specifying systems information needed to contact the source: the protocol (e.g. JDBC), the connection string, and a source-specific query driver. The driver translates RXL expressions into the source's query language, which is typically a dialect of SQL, but other query languages can be supported. For example, the executable RXL query in Fig. 7 is translated into the SQL query:

```
select  c.pid as pid, c.item as item
from    Clothing c, SalePrice s
where   c.category = "outerwear",
        c.pid = s.pid,
        s.price < 0.5 * c.retail
sort by c.pid
```

and into the XML template:

8

```
<results>
   <result ID=Result("Acme Clothing")>
        <supplier> "Acme Clothing" </supplier>
        <name ID=Name($pid, $item)>$item</name>
   </result>
</results>
```

where the variables `$pid` and `$item` refer to the attributes `pid` and `item` in the SQL query's `select` clause; we describe the template generation in more detail in Sec. 3.1. After translation, the SQL query is sent to the relational engine, and the resulting tuple stream is fed into the XML generator, which produces the XML output.

In this example, the translation requires only one SQL query. In general, there may be several ways to translate a complex RXL query into one or more SQL queries and to merge tuple streams into the XML result. Choosing an efficient evaluation strategy may be important when the RXL query returns a large result, e.g., if the entire XML view if materialized. Currently, SilkRoute has one evaluation strategy, which generates one SQL query for each disjunct of a RXL sub-query, which must be in disjunctive-normal form (DNF). Each SQL query has a `sort by` clause, making it possible for the XML generator to merge them into an XML document in a single pass.

2.5   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 precompute 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, precomputed 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 XML-QL, 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.

3   Query Composition

In this section, we describe the query composition algorithm. Recall that an RXL query, V, takes as input a relational database and returns as output an XML document. The XML-QL user query, U, is written against V; it takes as input an XML document and returns an XML document. For any database D, we can compute the result of U by first materializing $V(D)$, denoted as $XMLD$, and then computing $U(XMLD)$. The query composition problem is to construct an equivalent RXL query C, where $C = U \circ V$. In other words, we would like to construct an RXL query C that is guaranteed to yield the same result as U and V for any database D, that is, $C(D) = U(V(D))$. Notice that C takes as input a relational database and returns an XML document. With C, we skip the construction of the intermediate result $XMLD$.   As a

```
construct
<supplier ID=Supp()>
    <company ID=Comp()>"Acme Clothing"</company>
    {
      from  Clothing $c
      where $c.category = "outerwear"
      construct
        <product ID=Prod($c.pid)>
        <name   ID=Name($c.pid,$c.item)>$c.item</name>
        <category ID=Cat($c.pid,$c.category)>$c.category</category>
        <retail ID=Retail($c.pid,$c.price)>$c.price</retail>
          { from SalePrice $s
            where $s.pid = $c.pid
            construct
              <sale ID=Sale($c.pid,$s.pid,$s.price)>$s.price</sale>
          }
          { from Problems $p
            where $p.pid = $c.pid
            construct
            <report code=$p.code ID=Prob($c.pid,$p.pid,$p.code,$p.comments)>
              $p.comments
            </report>
          }
        </product>
    }
</supplier>
```

Figure 8: RXL view query (V) with patterns from XML-QL query (U) highlighted.

running example, we use V from Fig. 5, and U from Fig. 6; the result of the composition, C, is shown in Fig. 7.

Before describing the details, we give a brief intuitive description. An important observation is that all XML components (tags, attributes, #PCDATA) present in *XMLD* are explicitly mentioned in the construct clause(s) of V. When U is evaluated on *XMLD* its patterns are matched with these components. The key idea is then to evaluate U on V's templates directly, without constructing *XMLD*. During this evaluation we only consider the patterns, not the filters occurring in U. In our example, U has a unique pattern that mentions <supplier>, <company>, <product>, <name>, <retail>, and <sale> with a particular nesting, and all these tags indeed occur in V's templates under the same nesting. We show V again in Fig. 8, after the matching, with the matched tags in boxes. Once the matching is done, we construct the composed query C in a second step, as follows. C's construct clause is the same as U's construct clause, modulo variable renaming. Its from and where clauses consist both of the "relevant" from and where clauses in V and of all the where filter conditions in U, modulo variable renaming. This completes the construction of C. In our example, the "relevant" from and where clauses are (see Fig. 8):

```
from Clothing $c, SalePrice $s
where $c.category = "outerwear", $s.pid = $c.pid
```

and the where filter condition from U is `$sale < 0.5 * $retail` becomes the following after variable renaming:

```
where $s.price < 0.5 * $c.retail
```

The reader may check that, together, they form the from and where clauses in Fig. 7.

Fig. 9 depicts the architecture of query composition. The *pattern matcher* implements the first step; it evaluates U on V by matching U's patterns with V's templates. The result is a *solutions relation*, R, in which each tuple represents one match. Multiple matches may occur if the patterns contain alternation, e.g., <company|organization>, or Kleene-star operators, e.g., <*.supplier>, or tag variables <$elm>. The second step is implemented by the *rewriter*. It takes the remaining clauses (V's from and where and U's construct) and the relation R, and rewrites each solution tuple into one RXL block. The result is the composed query C.
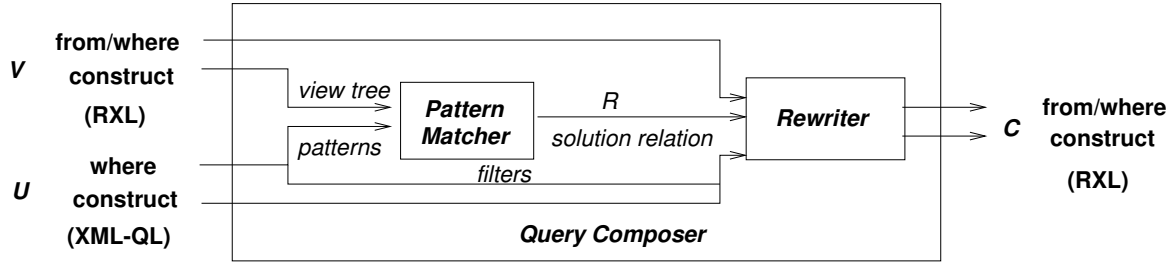
10

Figure 9: Diagram of Query Composition.

```
<supplier ID=Supp()>                              Supp()  :-    true
  <company ID=Comp()>Acme Clothing</company>      Comp()  :-    true
  <product ID=Prod($cpid)>                        Prod($cpid) :- Clothing($cpid, _, $category, _, _),
                                                                   $category = "outerwear"
    <name ID=Name($cpid,$citem)>                  Name($cpid, $citem) :- Clothing($cpid, $citem, c$ategory, _, _),
      $citem                                                              $category = "outerwear"
    </name>
    <category ID=Cat($cpid,$ccategory)>           Cat($cpid, $ccategory) :- Clothing($cpid, _, $category, _, _),
      $ccategory                                                            $category = "outerwear"
    </category>
    <retail ID=Retail($cpid,$cprice)>             Retail($cpid, $cprice) :-  Clothing($cpid, _, $category, _, $cprice),
      $cprice                                                                $category = "outerwear"
    </retail>
    <sale ID=Sale($cpid,$spid,$sprice)>           Sale($cpid, $spid, $sprice) :-
      $sprice                                              Clothing($cpid, _, $category, _, _), $category = "outerwear",
    </sale>                                                SalePrice($spid, $sprice), $cpid = $spid
    <report ID=Prob($cpid,$ppid,$pcode,$pcmnts)   Rep($cpid, $ppid, $pcode, $pcmnts) :-
          code=$pcode>                                     Clothing($cpid, _, $category, _, _), $category = "outerwear",
      $pcmnts                                              Problems($ppid, $pcode, $pcmnts), $cpid = $ppid
    </report>
  </product>
</supplier>
```

Figure 10: View tree for the RXL query in Fig. 5: template (left) and datalog rules (right).

We notice that our query composition technique can be viewed as an example of *partial evaluation*: the patterns are evaluated at *composition time* (a.k.a. compile time) on V's templates, and the filters and constructors are evaluated *at run time* when the new RXL view is evaluated. The remainder of this section describes the internal representation of view and user queries and gives a detailed description of the composition algorithm. A pseudo-code version of the algorithm appears in Appendix A.

3.1 Step 1: Pattern Matching

In Step 1, we construct the solutions relation R that contains all matchings of U's patterns with V's templates.

Construct the view tree. For the composition algorithm, we represent V by a data structure called a *view tree*, which consists of a global template and a set of datalog rules. The global template is obtained by merging all V's templates from all its construct clauses: nodes from two different templates are merged if and only if they have the same Skolem function, hence each Skolem function occurs exactly once in the view tree. The datalog rules are non-recursive. Their heads are the Skolem functions names, and their bodies consist of relation names and filters. The datalog rules are constructed as follows. For each occurrence of a Skolem function F in V, we construct one rule of the form $F(x, y, \ldots) : -body$, where *body* is the conjunction of all from and where clauses in the scope where F occurs. When a rule is associated with a Skolem function, we say that the rule *guards* that function and its corresponding XML element. In both the template and datalog rules, we replace the tuple variables used in RXL by column variables.

11

```
<supplier ID=$t1>
  <company ID=$t2>$company</company>
    <product ID=$t3>
      <name ID=$t4>$name</name>
      <retail ID=$t5>$retail</retail>
      <sale ID=$t6>$sale</sale>
    </product>
</supplier>
```

Figure 11: Adding one temporary variable for each element in the pattern.

Fig. 10 contains the view tree for our example. The unique `supplier` element is guarded by the rule `Supp() :- true`, which is always true, because no predicate expression guards the element's creation. The `retail` elements are guarded by the rule:

```
Retail($cpid, $cprice) :- Clothing($cpid, _, $category, _, $cprice), $category = "outerwear"
```

which means that one `retail` element is created for each value of `cpid` and `cprice` that satisfies the table expression on the right-hand side. There is only one datalog rule for each Skolem function, because each function occurs once in V (Fig. 5).

Evaluate U on the view tree. Next, we match U's patterns with V's template. To simplify presentation, we assume that U consists of a single block:

$$U = \texttt{construct <elm> \{ where } P, W \texttt{ construct } T \texttt{ \} </elm>} \tag{1}$$

where T denotes the template, P denotes all patterns, and W denotes all filters. For each element in U's patterns, we introduce one new, temporary variable for the element's `ID` attribute. In our example, U has a single pattern and we add six new variables shown in Fig. 11. We discuss the necessity of these variables and how to handle multi-block user queries in Sec. 3.3.

Next, we evaluate U's patterns on V's template in the standard way of evaluating patterns on a tree [9]. In general, there may be zero, one, or more results, and we represent them as a table R, with one column for each variable in U, and one row for each result. The values in the table are #PCDATA, Skolem terms, variables, tag names, attribute values, and attribute names, which occur in V's template. In our example, this step results in the following table R:

$t1	$t2	$company	$t3	$t4	$name	$t5	$retail	$t6	$sale
Supp()	Comp()	Acme Clothing	Prod($cpid)	Name($cpid, $cprice)	$citem	Retail($cpid, $cprice)	$cprice	Sale($cpid, $spid, $sprice)	$sprice

The column names correspond to the variables in U's pattern in Fig. 11. The single row in R means that there exists only one matching of U's pattern with V's template. The row specifies that U's variable `$name` is bound to `$citem` in V, the variable `$t3` is bound to the Skolem term `Prod($cpid)`, and the variable `$company` is bound to the text data `Acme Clothing`.

3.2 Step 2: Query Rewriting

In Step 2, we use the table R to construct the composed query C. Each row in R represents one match, and C is the union of all possible matches. In particular, C consists of several parallel blocks, which denote union in RXL. In each block, the `from` and `where` clauses contain the "relevant" datalog rules, which are the rules for the Skolem functions in the corresponding row. The block's `construct` clause contains U's template. Recall that U consists of a single block, Eq. 1, and that T denotes its template, P its patterns, and W its filters. Let the rows in R be $r_1 \ldots r_k$. Then C consists of several parallel blocks:

$$C = \texttt{construct \{ <elm>} \{B_1\} \ldots \{B_k\} \texttt{ </elm> \}}$$

with one or more blocks corresponding to each row r_i. Next, we show how to construct the blocks corresponding to one row, r_i, in R.

Construct one block. We first represent one block's `from` and `where` clauses as one datalog rule. Let $F_1 \ldots F_n$ be the Skolem functions that occur in the row r_i. Recall that the view tree associates one or more datalog rules to each Skolem function. Assume that there is a unique datalog rule for each Skolem function:

$$F_1 : - \; body_1 \; \ldots \; F_n : - \; body_n.$$

The block's `construct` clause is $\mathcal{S}_0(T)$ where $\mathcal{S}_0$ is a variable substitution defined below. For each datalog rule F_i, we apply one variable substitution $\mathcal{S}_i$. The body of the new datalog rule is the union of all bodies after variable substitution, plus $\mathcal{S}_0(W)$. Thus, the new rule has the form:

$$Q(\mathcal{S}_0(x), \mathcal{S}_0(y), \ldots) : -\mathcal{S}_0(W), \mathcal{S}_1(body_1), \ldots \mathcal{S}_n(body_n)$$

where $x, y, \ldots$ are the variables in U's template T. Next, we minimize Q, and rewrite it as a `from-where` clause: all relation names appear in the `from` clause, and all filters in the `where` clause. This completes the construction of one block.

Variable substitutions. Next, we define the substitutions $\mathcal{S}_0$ and $\mathcal{S}_1 \ldots \mathcal{S}_n$. For all the datalog rules $F_1 \ldots F_n$, we construct the substitutions $\mathcal{S}_1 \ldots \mathcal{S}_n$ so that the expressions $\mathcal{S}_1(body_1) \ldots \mathcal{S}_n(body_n)$ all have distinct variables, with one exception. For every two columns t_j, t_k in R, where the variable t_j corresponds to an element that is the parent of t_k's element, all variables in $\mathcal{S}_j(F_j(\ldots))$ must be shared with $\mathcal{S}_k(F_k(\ldots))$. To compute $\mathcal{S}_0$, we apply the substitutions $\mathcal{S}_1 \ldots \mathcal{S}_n$ to the entire row r_i and drop all columns in r_i that correspond to the temporary variables `$t1` $\ldots$ `$t2`. The new row is $\mathcal{S}_0$, which maps U's variables to column variables, constants, and Skolem terms.

When there is more than one datalog rule per Skolem function, we must convert the resulting datalog program into disjunctive normal form, i.e., a disjunction of multiple conjunctive datalog rules, before generating the RXL blocks. For each conjunctive rule, we apply the construction above to obtain one block and take the union of all such blocks. In this case, we obtain more than one block for one row r_i.

In our example, table R has one row that contains the Skolem terms `Supp()`, `Comp()`, `Prod($cpid)`, `Name($cpid, $citem)`, `Retail($cpid, $cprice)`, and `Sale($cpid, $spid, $sprice)`. Their corresponding datalog rules are in from Fig. 10. Now we compute the substitutions $\mathcal{S}_1, \ldots, \mathcal{S}_6$ such that the rules have disjoint variables with the exception of variables that have parent/child relationships. In our example, the variable `$t3` is the parent of variables `$t4`, `$t5`, `$t6` (Fig. 11). Therefore the Skolem term `Prod($cpid)` must share the variable `$cpid` with that in `Name($cpid, $citem)`, `Retail($cpid,$cprice)`, and `Sale($cpid, $spid, $sprice)`. Otherwise, all variables must be distinct. The modified rules are:

```
Supp()                   :-  true
Comp()                   :-  true
Prod($cpid)              :-  Clothing($cpid, _, $category1, _, _), $category1 = "outerwear"
Name($cpid, $citem)      :-  Clothing($cpid, $citem, $category2, _, _), $category2 = "outerwear"
Retail($cpid, $cprice)   :-  Clothing($cpid, _, $category3, _, $cprice), $category3 = "outerwear"
Sale($cpid, $spid, $sprice):-  Clothing($cpid, _, $category4, _, _),
                             $category4 = "outerwear", SalePrice($spid, $sprice), $cpid = $spid
```

The substitution $\mathcal{S}_0$ is obtained directly from the table R, by dropping all columns corresponding to the new variables `$t1`, ..., `$t6`:

$$\mathcal{S}_0 = \quad$$

$company	$name	$retail	$sale
Acme Clothing	$citem	$cprice	$sprice

U's template T is in Fig. 6 and its filter W is `$sale < 0.5 * $retail`. Only the variables `$company` and `$name` occur in T, so we have to include $\mathcal{S}_0(\$company)$ and $\mathcal{S}_0(\$name)$ in the rule's head; *company*, however, is a constant, therefore the rule becomes:

```
Q($citem) :- Clothing($cpid, _, $category1, _, _), $category1 = "outerwear",
             Clothing($cpid, $citem, $category2, _, _), $category2 = "outerwear",
             Clothing($cpid, _, $category3, _, $cprice), $category3 = "outerwear",
             Clothing($cpid, _, $category4, _, _),
             $category4 = "outerwear", SalePrice($spid, $sprice), $cpid = $spid,
             $sprice < 0.5 * cprice
```

The last line is $\mathcal{S}_0(W)$. We minimize Q (see e.g. [1]) and obtain the equivalent query:

```
Q($citem) :- Clothing($cpid, _, $category3, _, $cprice), $category3 = "outerwear",
             SalePrice($spid, $sprice), $cpid = $spid,
             $sprice < 0.5 * cprice
```

Finally, we convert it into `from` and `where` clauses, and add a `construct` clause whose template is $\mathcal{S}_0(T)$:

```
from  Clothing($cpid, _, $category3, _, $cprice),
      SalePrice($spid, $sprice)
where $category3 = "outerwear", $cpid = $spid, $sprice < 0.5 * cprice
construct <result ID= Result("Acme Clothing")>
              <supplier>Acme Clothing</supplier>
              <name>$citem</name>
          </result>
```

Lastly, we replace column variables by tuple variables, and we obtain the single-block query C in Fig. 7.

3.3   Details

Our example illustrates the simplest cases of query composition. Here, we give several other examples, which illustrate more complex cases.

View tree for multi-block query.   Consider the two-block RXL query in Fig. 4.

```
<view ID=View()>                              View() :- true
  <product ID=Prod($name)>                    Prod($name) :- Clothing($name, _)
                                              Prod($name) :- Clearance($name, $ddisc), $ddisc > 50
    <name ID=Name($name)>                     Name($name) :- Clothing($name, _)
      $name                                   Name($name) :- Clearance($name,  $ddisc), $ddisc > 50
    </name>
    <price ID=Price($name, $cprice)>          Price($name,$cprice) :- Clothing($name, $cprice)
      $cprice
    </price>
    <discount ID=Discount($name, $ddisc)>     Discount($name, $ddisc) :- Clearance($name, $ddisc),
      $ddisc                                                             $ddisc > 50
    </discount>
  </product>
</view>
```

Its view-tree template is on the left and its datalog rules on the right. In the RXL query, the Skolem functions `Prod` and `Name` occur twice. In the view tree, each function has two corresponding datalog rules, but in the template, they occur once.

Multiple rows.   In general, R may contain multiple rows. To illustrate, we use the query V (in Fig. 10) and compose it with the following XML-QL user query U':

```
construct <results> {
  where <supplier.product.(retail | sale)>$val</> in "http://acme.com/products.xml"
  construct <price>$val</price>
} </results>
```

The regular expression `supplier.product.(retail | sale)` matches a `retail` or a `sale` element nested in a `product` element, which is nested in a `supplier` element. It is analogous to the XPath expression `/supplier/product/retail|sale`. There are two matches of U with V, which produce two rows in R:

$t1	$t2	$t3	$val
Supp()	Prod($cpid)	Retail($cpid,$cprice)	$cprice
Supp()	Prod($cpid)	Sale($cpid,$spid,$sprice)	$sprice

The temporary variables $t1, $t2, $t3 are for supplier, product, and retail | price, respectively. The composed query C has two blocks:

```
C =  construct <results> { B1 } { B2 } </results>
```

The relevant datalog rules for the first row are those for Supp, Prod and Retail (Fig. 10). No variables are renamed, because $t2 is the parent of $t3. The generated datalog rule after minimization is:

```
Q($cprice) :- Clothing($cpid, _, $category, _, $cprice), $category = "outerwear"
```

and it produces C's first block $B1$:

```
B1 =    from Clothing $c
        where $c.category="outerwear"
        construct <price>$c.price</price>
```

The relevant datalog rules for the second row are those for Supp, Prod, and Sale. As before, no variables are renamed, and the datalog rule is:

```
Q($sprice) :- Clothing($cpid, _, $category, _, _), $category="outerwear", SalePrice($spid, $sprice), $cpid = $spid
```

which produces C's second block $B2$:

```
B2 =    from Clothing $c, SalePrice $s
        where $c.category="outerwear", $c.pid = $s.pid
        construct <price>$s.price</price>
```

Adding template variables. The temporary variables added to U's patterns play an important role, as revealed by the next example. The query V, written directly with column variables, is:

```
V = construct <v ID=H()> { from T($x, $y)
                            construct <a ID = F($x)>
                                          <b ID = G($x, $y)> $y </b>
                                       </a>
                          } </v>
```

and we consider two XML-QL queries U, U':

```
U = construct <results> { where <v><a><b>$z1</b> <b>$z2</b></a></v>
                          construct <result><z1>$z1</z1>  <z2>$z2</z2></result>
             } </results>
U'= construct <results> { where <v> <a><b>$z1</b></a> <a><b>$z2</b></a> </v>
                          construct <result><z1>$z1</z1>  <z2>$z2</z2></result>
             } </results>
```

Both return pairs of <b> values, but the first query returns pairs where both <b>'s are nested in the same <a> element. Without temporary variables in U's patterns, the relation R would be the same for U and U'. After introducing the new variables, the two relations R have different column names, and as expected, they produce two distinct composed queries.

Renaming variables in datalog rules. Continuing the previous example, we illustrate the need for the substitutions $S_1, S_2, \ldots$. First, we build V's view tree:

```
<v ID=H()>                          H()        :- true
   <a ID = F($x)>                   F($x)      :- T($x, _)
       <b ID = G($x, $y)>$y</b>     G($x, $y) :- T($x, $y)
   </a>
</v>
```

We illustrate the composition with U'. We add five temporary variables and its pattern becomes:

```
<v ID=$t1> <a ID=$t2><b ID=$t3>$z1</b></a>
            <a ID=$t4><b ID=$t5>$z2</b></a>
</v>
```

Matching the pattern with the template produces one row in R:

$t1	$t2	$t3	$z1	$t4	$t5	$z2
H()	F($x)	G($x,$y)	$y	F($x)	G($x,$y)	$y

Intuitively the variable $y in the $z1 column is different from $y in the $z2 column, because they match different <b> elements, possibly in different <a> elements. This distinction is made precise by the renaming step. Thus, after variable substitution, the five relevant datalog rules become:

```
H()         :- true
F($x1)      :- T($x1, _)
G($x1, $y1):- T($x1, $y1)
F($x2)      :- T($x2, _)
G($x2, $y2):- T($x2, $y2)
```

and the composed query C, after query minimization, is:

```
construct <results> from T($x1, $y1), T($x2, $y2)
          construct <result><z1>$y1</z1>  <z2>$y2</z2></result>
          </results>
```

XML-QL queries with block structure. In general, U may have several blocks, both nested and parallel. For multi-block user queries, we construct a different table R for each block in U, in the same way in which the XML-QL query processor handles multiple blocks. Tables corresponding to parallel blocks are independent; for nested blocks, there is a distinct inner table that corresponds to each row in the outer table. The composed query C follows the same block structure, except that one block in U may generate multiple parallel blocks in C, as described earlier in this section.

Query minimization. Query minimization eliminates redundancies in queries, such as duplicate conditions. Query minimization can be expensive, because it is NP-complete. Commercial database systems often do not perform minimization, because users typically do not write redundant queries. In SilkRoute, the composed query C is generated automatically. One condition in a view query V may appear in multiple datalog rules, and, hence be propagated as multiple copies in the generated query C. To avoid query minimization, one could trace these repetitions to the original RXL query, but care is needed to deal with variable renaming. For RXL queries with large parallel blocks, however, query minimization may be unavoidable.

3.4   Aggregation Queries

We briefly describe how aggregations in XML-QL queries can be "pushed" into composed RXL views and evaluated by the target RDBMs. In both XML-QL and RXL, we use Skolem terms to specify the values by which aggregate expressions are grouped.

Suppose a reseller wants to count the total number of reports for each defective product. This can be expressed in XML-QL as follows:

```
where <supplier.product ID=$pid>
        <name>$n</>
        <report>$r</>
      </> in "http://acme.com/products.xml"
construct <product ID=F($pid)>
           <name ID=G($pid,$n)>$n</>
           <totaldefects ID=H($pid)>count(*)</>
          </>
```

The Skolem term `F($pid)` in `<product ID=F($pid)>` asserts that all bindings for the variables `$pid`, `$n` and `$r` are grouped by `$pid`'s value. Similarly, the Skolem term `H($pid)` specifies the grouping attributes for the aggregate function `count(*)`, which counts the total number of bindings. This idea is similar to the group-by construct in SQL. XML-QL and RXL's semantics guarantee that only one element is produced for each value of a Skolem term, e.g., one `name` element is emitted for each value of `$n`.

We use a simple extension to datalog that accommodates aggregate functions [1]. An example of a datalog rule that uses a "generator" to count values is:

```
C(p, q, COUNT(*)) :- R(p, q)
```

Only the last argument in the head can be an aggregate function; the other arguments specify the grouping attributes. The meaning is that C contains the set of triples (p, q, r) where r is the number of tuples in the group corresponding to values (p, q) in the relation R.

Using our composition algorithm, the XML-QL query above can be rewritten as:

```
from  Clothing $c, Problems $p
where $c.pid = $p.pid
construct <product ID=F($c.pid)>
            <name ID=G($c.pid, $c.item)>$c.item</>
            <totaldefects ID=H($c.pid)>count(*)</>
         </product>
```

Note that the aggregate function is "pushed" into the RXL view. When this view is materialized, the aggregation will be evaluated by the relational engine. Most importantly, this query can be evaluated efficiently, because commercial database systems are often highly optimized for aggregation queries.

4   Related Research and Systems

Bosworth [3] discusses the need for tools that export relational data into XML views. Several commercial tools already exist. The ODBC2XML tool allows users to define XML documents with embedded SQL statements, which allows them to construct an XML view of the relational data. Such views are materialized, however, and cannot be further queried with an XML query language like XML-QL. Alternatively, Oracle's XSQL tool defines a fixed, canonical mapping of the relational data into an XML document, by mapping each relation and attribute name to an XML tag and tuples as nested elements. Such a view could be kept virtual, but the approach is not general enough to support mappings into arbitrary XML. IBM's DB2 XML Extender provides a Data Access Definition (DAD) language that supports both composition of relational data in XML and decomposition of XML data into relational tables. DAD's composition feature, like RXL, supports generation of arbitrary XML from relational data. Unlike RXL, the criteria for grouping elements is implicit in the DAD and DAD specifications cannot be nested arbitrarily. More significantly, XML Extender does not support query composition, however, DAD could be used as a view-definition language in a SilkRoute architecture.

XML-QL was first proposed as a W3C technical note [8], then published in [9]. It is the first complete query language specifically designed for XML, borrowing ideas from the research area on semistructured data [2, 4, 11]. Other query languages for XML include XSL [5, 6], XQL [20], Lorel [15], XMAS [13], and YaTL [7].

Query composition is simple for select-project-join queries [1, 19], and for the relational calculus [1], in general. In the context of semistructured data, Papakonstantinou et al. first address the problem in the framework of MSL [16], a datalog-like language. Their composition algorithm, called *query decomposition and algebraic optimization*, uses a unification algorithm on the view's head and the query's body. Deutsch et al. [10] and Papakonstantinou and Vassalos [18] address query composition in the more complex setting of query rewriting for semistructured data. Our solution borrows ideas from [10].

5 Discussion

SilkRoute is a general, dynamic, and efficient framework for viewing and querying relational data in XML. We believe it is the first XML-export tool to support arbitrarily complex, virtual views of relational data and to support XML user queries over virtual views. The ability to support arbitrary views is critical in data exchange between inter-enterprise applications, which must abide by public DTDs and cannot reveal the underlying schemas of their proprietary data. The main contribution of this work is a sound and complete algorithm for composing virtual views and user queries. SilkRoute has many benefits: only the relational data requested by a user query is ever materialized; that data is always produced on demand; and the relational engine performs most of the computation efficiently.

We have not addressed two important open problems: general techniques for translating of RXL into efficient SQL and minimization of composed RXL views. As discussed in Sec. 2.4, SilkRoute has one translation strategy, which generates one SQL query for each RXL sub query, which must be in disjunctive-normal form (DNF). In practice, RXL view queries can be arbitrary boolean combinations of table and filter expressions; for example, parallel RXL blocks often construct parts of complex elements independently, i.e., they express unions. User queries over such views often produce composed queries with many unions. Currently, we normalize any RXL sub-query into multiple sub-queries in DNF, which is a quadratic increase in the number of sub-queries to evaluate. In practice, we may be able to translate multiple queries in DNF directly into SQL, for example, by using SQL's union-join constructs. Similarly, nested RXL queries often express left outer joins, e.g., the parent sub-query is the left relation and the child sub-query is the right relation. Currently, we generate two SQL queries, one for parent and child, but clearly one SQL query suffices. In addition to reducing the number of SQL queries, we want to minimize each individual RXL sub-query, i.e., eliminate all redundant expressions, so that the resulting SQL query is also minimal. Techniques exist for query minimization [1], but general algorithms are NP-complete. We expect heuristic algorithms to be effective for RXL queries, because RXL's nested block structure can help identify those expressions that most likely are redundant. Our future research will focus on these problems and on applying SilkRoute to large-scale applications.

Availability. XML-QL and SilkRoute are implemented in Java. SilkRoute has drivers for Oracle and MySQL database servers. XML-QL can be downloaded from http://www.research.att.com/sw/tools/xmlql. SilkRoute should be available publicly in 2000.

A Composition Algorithm – Pseudocode

In the formal description of the algorithm, we need a notation for describing the types of values that are manipulated, e.g., view trees, XML-QL blocks. We denote types by grammar rules, like the following:

```
Node       :- Tag, Rule, [ Node ]
Rule       :- SkolemTerm, [ Condition ]
Condition :- TableExpr(String, [ Var ])
           | Filter
           | Or([ Condition ], [ Condition ])
Filter     :- And(Filter, Filter) | Or(Filter, Filter) | Not(Filter) | Term RelOp Term
```

These rules specify that a view tree `Node` is composed of a tag, a rule, and a list of children nodes; a `Rule` is composed of a Skolem term (its head) and a conjunctive list of conditions (its body); and a `Condition` is either a table expression, a filter expression, or the disjunction of two lists of conjuncts.

An XML-QL block is represented by a list of patterns, a list of filters, and a template, and an RXL block by a list of conditions and a template:

```
XMLQL   :- [ Pattern ], [ BoolExpr ], Template
RXL     :- [ Condition ], Template
```

```
 1. // Top-level invocation of compose function
 2. X_env = new [("$viewtree", Root())]
 3. S = new []
 4. R_block_list = compose(X_env , S, X_block)
 5.
 6. fun compose(Env X_env, VarMap S, XMLQL X_block) : [ RXL ] {
 7.   (X_patterns, X_filters, X_template) = decompose(X_block);
 8.
 9.   // Get pairs of (parent, child) variables from XML-QL patterns
10.   X_parent_child_vars = getHeadTargetMap(X_patterns);
11.
12.   // Evaluate pattern on view tree
13.   R = evalPattern(X_patterns, X_env);
14.
15.   // Consider each potential solution
16.   R_blocks = new []
17.   for each r_i in R {
18.     // Extend current environment with new variable bindings
19.     X_env' = appendList(X_env, r_i);
20.
21.     // Compute new S variable substitution from X_nodemap
22.     S' = newVariables(X_env', X_parent_child_vars, S);
23.
24.     // Compute RXL block for potential solution
25.     R_blocks = listAppend(oneSolution(X_env', S', X_block, r_i), R_blocks)
26.   }
27.   return R_blocks
28. }
```

Figure 12: Composition algorithm : top-level `compose` function

A template is either: a constant string; a variable; an element, which includes a tag and list of nested templates; or a nested query. To simplify presentation, templates are polymorphic, i.e., an XML-QL template contains only a nested XML-QL block and similarly, for an RXL template.

```
Template :- Const(String)
          | Var(String)
          | Element(Tag, [ Template ])
          | NestedQuery(XMLQL)
          | NestedQuery(RXL)
```

Finally, a canonical pattern is represented by the head variable (that occurs on the right-hand side of in), a regular-path expression over strings, and the target variable (that occurs in the body of an element):

```
Pattern :- Var, RegPE, Var
RegPE   :- String | Concat(RegPE, RegPE) | Alt(RegPE, RegPE) | Star(RegPE)
```

Regular-path expressions can be combined with the concatenation (.), alternation ($|$), and Kleene-star ($*$) operators, similar to those used in regular expressions.

The composition function `compose` in Fig. 12 takes two *environments*, which are lists of (XML-QL variable, value) pairs. The initial environment (`X_env`) maps the distinguished variable `$viewtree` to the root of the view tree referenced by the user query U. The initial variable-substitution `S` that maps XML-QL variables to RXL expressions is empty, and `X_block` is the top-level XML-QL block (lines 1–3). In our example, `$viewtree` is bound to the root of the tree in Fig. 10. The result of `compose` is a list of RXL blocks. In the pseudo-code, XML-QL expressions are prefixed by `X_` and RXL expressions by `R_`.

Function `compose` (line 7) decomposes `X_block` into its patterns, filters, and template. New temporary variables are introduced to represent the intermediate nodes in the nested pattern.

On line 13, the patterns are evaluated in the current environment, producing `R`, which maps U's variables to nodes and constants in V. Each tuple in `R` represents one possible rewriting of U over the view. For each tuple `r_i`, the current environment is extended with the new variable bindings (line 19).

Function `newVariables` (line 22) computes the new mappings of XML-QL and RXL variables to common RXL variables. In summary, `newVariables` recovers the correspondence between Skolem terms that share a

19

```
1.  // Return new RXL block for potential solution in r_i
2.  fun oneSolution(Env X_env, VarMap S, XMLQL X_block, Env r_i) : [ RXL ] {
3.     R_conditions = new []
4.     // For each XML-QL variable X_v in X_block
5.     foreach X_v in getVariables(X_block) {
6.       // Get view-tree node bound to X_v
7.       R_node = project(r_i, X_v);
8.       // Get rule associated with view-tree node
9.       (R_tag, R_rule, R_children) = R_node
10.      // Get body of rule
11.      (R_head, R_body) = R_node;
12.      foreach R_condition in R_body {
13.        R_condition' = makeCopy(R_condition)
14.        // Rename head variables in R_condition' and add to R_conditions
15.        R_conditions = cons(rewriteR(S, R_condition'), R_conditions)
16.      }
17.    }
18.    // Rename variables in X_filters and add to R_conditions
19.    foreach X_filter in X_filters
20.      R_conditions = cons(rewriteX(X_env, S, X_filter), R_conditions)
21.
22.    // Put conditions in disjunctive normal form, i.e., [[ Condition ]]
23.    R_disjuncts = to_DNF(R_conditions)
24.
25.    // Rename variables in X_template
26.    R_template = rewriteX(X_env, S, X_template)
27.
28.    R_blocks = []
29.    // Construct new RXL block: solution conditions + RXL template
30.    foreach R_conjunct in R_disjuncts
31.      R_blocks = cons(new RXL(R_conjunct, R_template), R_blocks)
32.
33.    return RXL_blocks
34. }
```

Figure 13: Composition algorithm: `oneSolution` function

common ancestor in the XML-QL pattern; this correspondences determines the mappings for RXL variables. For XML-QL variables, the mapping is simple: if the corresponding value is a leaf node or constant value, the variable is replaced by its value in the substitition mapping $\mathcal{S}$, as described in Section 3.2. If the corresponding value is an internal node, the variable is replaced by the complete RXL expression that computes that element under the substitution $\mathcal{S}$. Lastly, function `oneSolution` (line 25) takes the new environment and computes the new RXL blocks, which are appended to the list of other potential solutions.

Function `oneSolution` in Fig. 13 constructs the RXL block(s) in four steps. First, for each XML-QL variable `X_v` in `X_block`, it projects `X_v`'s value from the solution tuple `r_i`; its value is a view-tree element and an associated rule, whose head and body are projected in `R_head` and `R_body` (a list of conditions). Function `makeCopy` (line 13) assigns fresh variable names to all free variables in `R_condition`, i.e., those that do not occur in the rule's head. Function `rewriteR` (line 15) rewrites the new rule, using the variable mapping `S`. The new condition is added to the conjunctive conditions in `R_conditions`. Second, the function `rewriteX` (line 20) rewrites the XML-QL filters in `X_filters` and adds those to `R_conditions`. Third, the function `to_DNF` (line 22) puts the new conditions in disjunctive normal form. On line 26, `rewriteX` rewrites the XML-QL template to produce the new RXL template. Finally, one new RXL block is created for each list of conjuncts in `R_disjuncts`, and the union of all these blocks is returned.

The `rewriteX` and `rewriteR` functions in Fig. 14 replace XML-QL and RXL variables by their new names in `S`. The "helper" functions `substX` and `substR` perform the variable substitutions. Note that `rewriteX` calls `compose` recursively to rewrite a nested XML-QL block into an equivalent nested RXL block.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley Publishing Co, 1995.

```
1. // rewriteX rewrites XML-QL expression as RXL expression
2. fun rewriteX(Env X_env, VarMap S, X_Expr E) {
3.    fun substX(E) {
4.       case E of
5.          Var(v)              = lookupMap(S, v)
6.          Const(c)            = new Const(c)
7.          Element(T, X)       = new Element(T, mapList(substX, X))
8.          Relop(op,E1,E2)     = new Relop(op, substX(E1), substX(E2))
9.          // Cases for all types of BoolExprs . . .
10.         // Recursively compose and rewrite nested XML-QL query
11.         NestedQuery(X_block) = new NestedQuery(compose(X_env, S, X_block))
12.      }
13.   return substX(E)
14. }
15. // rewriteR renames RXL variables.
16. fun rewriteR(S varmap, R_Expr E) {
17.    fun substR(E) {
18.       case E of
19.          Var(v)              = lookupMap(S, v)
20.          TableExpr(name, vars) = new TableExpr(name, mapList(substR, vars))
21.          Filter(b)           = new Filter(substR(b))
22.          Or(l1, l2)          = new Or(mapList(subst, l1), mapList(substR, l2))
23.          // Cases for all types of BoolExprs . . .
24.          NestedQuery(RXL(conditions, template)) =
25.            new NestedQuery(new RXL(mapList(substR, conditions), substR template))
26.      }
27.   return substR(E)
28. }
```

Figure 14: Composition Algorithm: `rewrite` function

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[3] A. Bosworth and A. L. Brown. Microsoft's vision for XML. *IEEE Data Engineering Bulletin*, pages 37–45, Sept 1999.

[4] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[5] J. Clark. XML path language (XPath), 1999. `http://www.w3.org/TR/xpath`.

[6] J. Clark. XSL transformations (XSLT) specification, 1999. `http://www.w3.org/TR/WD-xslt`.

[7] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion ! In *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 177–188, 1998.

[8] A. Deutsch, M. Fernńdez, D. Florescu, A. Levy, and D. Suciu. Xml-ql: A query language for XML, 1998. `http://www.w3.org/TR/NOTE-xml-ql/`.

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

[10] A. Deutsch, M. Fernńdez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1999.

[11] M. Fernńdez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experience with a web-site management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.

[12] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of VLDB*, Athens, Greece, 1997.

[13] B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View definition and DTD inference for xml. In *Workshop on Semistructured Data and Nonstandard Data Formats*, January 1999.

[14] D. Maier. A logic for objects. In *Proceedings of Workshop on Deductive Database and Logic Programming*, Washington, D.C., August 1986.

[15] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, Edinburgh, UK, September 1999.

[16] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of Very Large Data Bases*, pages 413–424, September 1996.

[17] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[18] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 455–466, Philadephia, PA, June 1999. ACM Press.

[19] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 2nd edition, 1999.

[20] J. Robie. The design of XQL, 1999. `http://www.texcel.no/whitepapers/xql-design.html`.

[21] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.

Vitae. Mary Fernández is Senior Technical Staff Member in Large-Scale Programming Research at AT&T Labs. Dan Suciu is Principal Technical Staff Member in Information Systems and Analysis Research at AT&T Labs. Wang-Chiew Tan is a Ph.D. candidate in Computer Science at University of Pennsylvania.