Efficient Evaluation of XML Middle-ware Queries

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

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

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

ABSTRACT

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

1. INTRODUCTION

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

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

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

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

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

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

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

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

Figure 1: Fragment of TPC-H Schema

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

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

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

2. MOTIVATING EXAMPLE

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

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

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

Figure 2: DTD of XML data about suppliers.

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

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

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

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

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

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

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

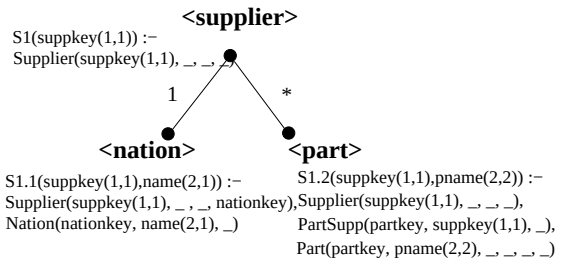


Figure 4: View tree for query fragment

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

The set of all possible choices are best visualized on the intermediate representation for RXL queries, which we call a *view tree*. Fig. 4 depicts the view tree for our simplified RXL query. Each node corresponds to an element in one of the `construct` clauses in the RXL query, and is annotated by a non-recursive datalog query that computes all instances of that node in the output XML. It is also possible to derive from the queries the multiplicities of the parent/child relationships, which are indicated by the labels `1` and `*`: the `1` between `<supplier>` and `<nation>` indicates that each `<supplier>` element in the output XML document will have exactly one child of type `<nation>`, and the `*` between `<supplier>` and `<part>` means that `<supplier>` may have zero or more many children of type `<part>`. Sec. 3.5 describes how to derive the multiplicities automatically.

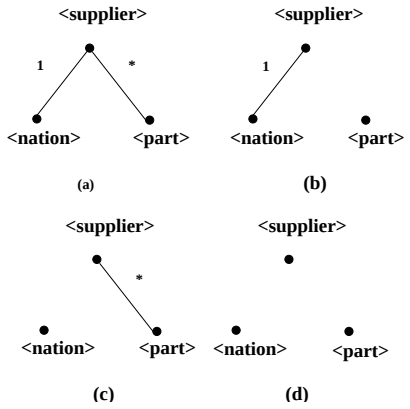


Figure 5: Execution plans for query fragment

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

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

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

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

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

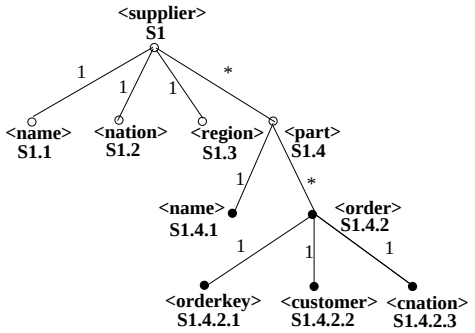


Figure 6: Labeled view tree for Query 1

```
order  by s.suppkey

select s.suppkey, p.name
from   Supplier s, Part p, PartSupp ps
where  s.suppkey = ps.suppkey and
       ps.partkey = p.partkey
order by s.suppkey
```

Notice that no outer join is needed, because the first query produces all the values for `Supplier`. The tagger must merge the two sorted tuple streams to produce the XML elements. Fig. 5(c) corresponds to two SQL queries and Fig. 5(d) to three SQL queries (omitted).

Fig. 6 depicts the view tree for the large RXL query in Fig. 3. In this view tree, there are nine edges and 2^9 or 512 subsets of edges, each of which corresponds to a partition of the tree. Therefore there are 512 possible plans for splitting the tree into a collection of SQL queries; each plan consists of between 1 and 10 tuple streams. On a TPC/H database of 100MB, some running times were:

No. of queries	Total Time	Query Time
10	1837s	584s
5	592s	244s
1	2729s	1234s

Total time includes the time to execute the query at the server and to bind and transfer the data to SilkRoute. The first plan splits the RXL query into ten small SQL queries, whose sorted tuple streams are merged by the tagger. The best plan consists of five SQL queries; in this case, the tagger merges five tuple streams. The last plan is the sorted outer-union plan. We note here that the optimal plan is substantially faster than the fully partitioned and sorted outer union plans, both of which one might expect to perform well. Also, several other plans, consisting of 3, 4, and 6 SQL queries, performed almost as well as the optimal plan: under 600s (246s).

In general, there are $2^{|E|}$ possible translations of an RXL query into one or more SQL queries, where $|E|$ is the number of edges in the query's corresponding view tree. Given the exponential number of potential plans, SilkRoute uses heuristics to choose a good plan. In commercial XML middle-ware products, the user typically must write these SQL queries himself, which effectively "hard wires" the evaluation plan into the XML view. This may seem like a reasonable requirement, but in practice, it is difficult to choose a good plan. The simplest choices are to always produce one unified relation as in Fig. 5(a) or fully partitioned relations as in Fig. 5(d). We show in Sec. 4 that these plans are often substantially slower than the optimal plans.

3. PLAN GENERATION

This section gives a formal definition of a view tree and describes the algorithm for translating a partitioned view tree into one or more SQL queries. Fig. 7 depicts the architecture of the planner and translator. The planner partitions a view tree into one or more subtrees; for each subtree, one SQL query is generated. The translator submits the SQL queries to the underlying RDBMS, reads in the result relations, and constructs one integrated (logical) relation. A tuple in the integrated relation represents a path from the root element to a leaf element in the result XML document. The XML document is constructed by re-nesting the tuples in the result relation and tagging each element.

3.1 View tree

We represent an RXL view query V by a *view tree*, which consists of a global XML template and a set of datalog rules. The global XML template is obtained by merging all V's XML templates from all its `construct` clauses. Every XML template has an associated Skolem term, that uniquely identifies the XML template in an RXL view. Elements from two different XML templates are merged if and only if they have the same Skolem function, hence each Skolem function occurs exactly once in the view tree. For example, the tree in Fig. 4 represents the global XML template for the query fragment in Fig. 3. Users can define Skolem terms explicitly in each XML element, in order to control how elements are grouped. Where Skolem terms are missing, the system introduces them automatically: it creates a new Skolem function name for each element; and its arguments are all keys of all tuple variables whose scope includes the XML element and all variables that are contained in that element. In Fig. 4, the system introduced Skolem functions S1, S1.1, and S1.2; the argument of S1 is $\text{suppkey}_{(1,1)}$, thus the Skolem term S1($\text{suppkey}_{(1,1)}$) uniquely identifies the `supplier` element. SilkRoute's XML generator uses the XML template to instantiate the result document. The arguments of S1.1 should be $\text{suppkey}_{(1,1)}$, nationkey, and $\text{name}_{(2,1)}$, and S1.2's arguments should be $\text{suppkey}_{(1,1)}$, partkey, and $\text{pname}_{(2,2)}$. To simplify presentation, we assume that name functionally determines nationkey, and pname functionally determines partkey, which simplifies S1.1's arguments to $\text{suppkey}_{(1,1)}$ and $\text{name}_{(2,1)}$, and S1.2's arguments to $\text{suppkey}_{(1,1)}$ and $\text{pname}_{(2,2)}$.

A view tree's datalog rules are non-recursive. Their heads are Skolem terms, 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 $F(x, y, \dots)$:- *body*, where *body* is the conjunction of all `from` and `where` clauses in the scope where F occurs. In both the XML template and in the datalog rules, we

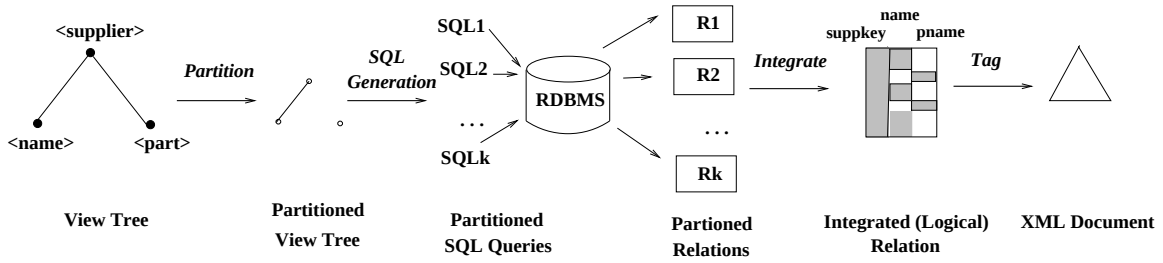


Figure 7: Architecture of query planner and translator

replace the tuple variables used in RXL by column variables. The head of a datalog rule corresponds to an element in the XML template, and a rule's body defines the conditions under which the element is created.

When assigning a Skolem term to a node, we associate a *Skolem-function index* with each Skolem function and a *Skolem-term variable index* with each Skolem variable. A Skolem-function index uniquely defines the tag and location of a node. These indices are used to sort the tuples of partitioned relations and during tagging of the XML document. An index $(l_1.l_2.\dots)$ is assigned to each node in breadth-first order. For example, $S1$ is assigned to the root, and $S1.1$ is assigned to the root's first child (Fig. 4 and 6). Each Skolem-term variable v is assigned a Skolem-term variable index (p, q) as follows. Let n_v be the node closest to the root that has v in its Skolem term. Then, p is equal to the level of n_v in the view tree, and q is the first integer that such that (p, q) is unique for all variables in the tree. For example, in Fig. 4 the variable `suppkey` is assigned index (1,1), because its containing element is at level one, and it is the first variable in the term, and the variable `pname` is assigned (2,2), because it is the second variable that appears in a term at level two.

3.2 View-tree partitioning

The planner produces one plan for each spanning forest of the view tree, so it produces $2^{|E|}$ plans, where $|E|$ is the number of view-tree edges. Fig. 5 contains the possible plans for the query fragment in Fig. 3. The planner produces one SQL query for each tree in a spanning forest. In Sec. 5, we present a greedy algorithm that heuristically chooses a subset of the $2^{|E|}$ plans.

For each tree in a spanning forest, we must define the schema of the relation that computes the nodes in the tree. Consider the unified plan in Fig. 5(a) that corresponds to the entire view tree. Given the database instance on the left in Fig. 8, the corresponding query produces the XML document on the right. The result of the SQL query is the relation in Fig. 9. Note that this relation is equivalent to the integrated relation.

In general, let T_i be one spanning tree in a partitioned view tree T, and let $SFImax(T_i)$ be the maximum length of the Skolem-function indices in T_i. Let R_i be the partitioned relation that corresponds to T_i. Its schema is defined to be the set of attributes $attrs(R_i) = SFIattrs_i \cup STVattrs_i$ where

L_1	L_2	$s.suppkey_{(1,1)}$	$n.name_{(2,1)}$	$p.name_{(2,2)}$
1	1	supp#1	USA	
1	2	supp#1		plated brass
1	2	supp#1		anodized steel
1	1	supp#2	Spain	
1	1	supp#3	France	
1	2	supp#3		polished nickel

Figure 9: Integrated relation for Plan (a)

Plan (c), node `<nation>`;

L_1	L_2	$suppkey_{(1,1)}$	$name_{(2,1)}$
1	1	supp#1	USA
1	1	supp#2	Spain
1	1	supp#3	France

Plan (c), edge `<supplier>-<part>`:

L_1	L_2	$suppkey_{(1,1)}$	$pname_{(2,2)}$
1	2	supp#1	plated brass
1	2	supp#1	anodized steel
1		supp#2	
1	2	supp#3	polished nickel

Figure 10: Relations for Plan (c) in Fig. 5

- $SFIattrs_i = \{"L_j" | 1 \le j \le SFImax(T_i)\}$, and
- $STVattrs_i = \{v | v,$ a Skolem-term variable in $T_i\}$.

Fig. 10 contains partitioned relations for Fig. 5(c). The first relation corresponds to the tree containing only the `nation` node: its Skolem-function index contains the labels L_1 and L_2, and two Skolem-term variables $\text{suppkey}_{(1,1)}$ and $\text{name}_{(2,1)}$. The tuples in an instance of a partitioned relation R_i are sorted by $L_1, V_{(1,1)} \dots, V_{(1,n_1)}$, $L_2, V_{(2,1)} \dots, V_{(2,n_2)}$, etc. This order is consistent with the structural relationship between the elements in the result XML document.

3.3 Integration and tagging

Regardless of the number of trees into which we partition the view tree, the associated partitioned relations R_i are sufficient to reconstruct the integrated relation (corresponding to a single partition of the viewtree), in a single pass. In SilkRoute, the integrated relation is virtual, i.e., it does not materialize the relation. Instead, the result XML document is constructed directly from the partitioned relations. Limited space prevents presentation of the tagging algorithm. Informally, the tagging algorithm merges the partitioned tuple streams into one tuple stream, nests the tuples, and tags their

```
Supplier(supp#1, "USA Metalworks",   "New York", usa#24)
Supplier(supp#2, "Romana Espanola",  "Madrid",   spain#3)
Supplier(supp#3, "Fonderie Francais","Paris",    france#19)

Nation(usa#24,  "USA",     reg#1)
Nation(japan#3, "Spain",   reg#2)
Nation(rom#19,  "France",  reg#3)

PartSupp(part#4,  supp#1, 100)
PartSupp(part#12, supp#1, 320)
PartSupp(part#20, supp#3,  64)

Part(part#4,  "plated brass",    mfgr#3, "Brand1", "S", 904.00)
Part(part#12, "anodized steel",  mfgr#4, "Brand2", "M", 912.01)
Part(part#20, "polished nickel", mfgr#1, "Brand3", "L", 920.02)
```

```
<supplier key="supp#1">
  <nation>USA</nation>
  <part>plated brass</part>
  <part>anodized steel</part>
</supplier>
<supplier key="supp#2">
  <nation>Spain</nation>
</supplier>
<supplier key="supp#3">
  <nation>France</nation>
  <part>polished nickel</part>
</supplier>
```

Figure 8: Example fragment of TPC-H database instance and fragment of result XML document

values. The required memory size of the algorithm depends only on the number of nodes and Skolem-term variables in the view tree. It does *not* depend on the size of the database instance, therefore the algorithm scales well as the size of the underlying database, and corresponding XML document, increases.

3.4 SQL generation

SilkRoute uses *outer-join* plans to construct queries for partitioned relations. The outer-join plans can be implemented using the outer-join and union operators of SQL. For example, one possible SQL query for Fig. 5(a) uses a left-outer join to combine the root (`supplier`) node with its children nodes, and it uses a outer union to combine the children nodes (the `nation` and `part` elements). [1]

```
select 1 as L1, L2, s.suppkey, Q.name, Q.pname
from Supplier s
left outer join
((select 1 as L2, n.nationkey as nationkey,
          n.name as name, null as suppkey,
          null as pname
  from  Nation n)
 union
 (select 2 as L2, null as nationkey,
         null as name, ps.suppkey as suppkey,
         p.name as pname
  from PartSupp ps, Part p
  where ps.partkey = p.partkey)
) as Q
on (L2=1 and s.nationkey = Q.nationkey)
    or (L2=2 and s.suppkey = Q.suppkey)
sort by L1, s.suppkey, L2, Q.nationkey,
        Q.name, Q.pname
```

The structure of outer-join plans using left-outer joins and unions corresponds closely to the structure of subtrees. The sub-query for a node n in a view tree and the sub-queries of n's children are combined with an outer join. The sub-queries for n's children (siblings) are combined with an outer union. The outer union is necessary because sibling nodes have different relational structures: in the relation that computes a node m, the attributes of m's siblings are null values. The query above can be simplified further by *view-tree reduction*, which we describe in the next section.

The outer-join plan is different from the outer-union plan [9] described in Sec. 2. They correspond to (R *left-join* $(S \cup T)$) and ((R *leftjoin* S) $\cup$ (R *leftjoin* T)), respectively. The outer-union plan combines parent and child nodes using inner or outer joins and combines subtress with outer unions. In general, SilkRoute can use either strategy to generate queries, but it currently implements outer-join plans. For completeness, we include the outer-union plan in the experiments in Sec. 4 and distinguish clearly between the unified outer-join and outer-union plans in our results.

Some of the plans SilkRoute produces do not require outer union, outer join, or the with clause. For example, a fully partitioned plan has no edges and requires none of these constructs. Plans with no branches (i.e., no sibling nodes) do not require the union operator. This characteristic is especially useful in a middle-ware system, because all SQL engines do not necessarily support all these constructs. In those cases, SilkRoute chooses permissible plans based on the source description of the underlying RDBMS.

3.5 View-tree reduction

The view tree is a flexible intermediate representation, because it supports generation of multiple execution plans, but its flexibility can introduce redundant queries in the view tree and in corresponding execution plans. Recall that a single condition in an RXL query often guards the creation of multiple elements, e.g., the `part` and `name` elements in Query 1 are both guarded by the conditions `$s.suppkey = $ps.suppkey` and `$ps.partkey = $p.partkey`. In the corresponding view tree, the two elements are guarded by distinct, but equivalent, datalog rules. During plan generation, we would like to eliminate redundant queries by iden-

[1] We also can use the SQL 'with' clause to construct partitioned relations that satisfy the definition in Section 4.3.1, if the RDBMS supports it.

tifying "reducible" edges in the view tree. An edge is reducible if the queries associated with its nodes are equivalent, or if the query associated with a child node has a functional or inclusion dependency on the query of its parent node. In both cases, one query can be eliminated, because it is implied by the other.

After generating a partitioned view tree, the planner reduces the view tree in two steps. First, edges in the view tree are assigned labels that indicate the potential number of child elements in the result XML instance. Second, groups of nodes connected by '1'-labeled edges, which represent functionally dependent queries, are collapsed into one node by combining their queries. After view-tree reduction, SQL generation proceeds as described in Sec. 3.4.

We illustrate the labeling step on the view tree in Fig. 6. The edge labels '1', '?', '+' and '*', denote one, zero or one, one or more, and zero or more child elements, respectively. An RXL query does not contain sufficient information to label edges, because the possible number of XML elements depends on the database constraints. Currently, the database constraints are specified in a source description file, but they could be derived from key constraints and referential constraints extracted from the schema of the target database. Given these inputs, SilkRoute labels the view tree edges as follows. Assume that p and c are the parent and child nodes of an edge e, where their queries are $F(x_1, \ldots, x_m)$:- Q_p and $G(x_1, \ldots, x_m, \ldots, x_n)$:- Q_c, respectively. Let $R_p = \{\langle x_1, \ldots, x_m\rangle | Q_p\}$ and $R_c = \{\langle x_1, \ldots, x_m, \ldots, x_n\rangle | Q_c\}$ be the relations defined by queries Q_p and Q_c. Then, e is labeled:

		C1	
		true	*false*
C2	*true*	1	+
	false	?	*

- **C1** is true if and only if there exists a functional dependency $R_c : x_1, \ldots, x_m \rightarrow x_{m+1}, \ldots, x_n$.
- **C2** is true if and only if there exists an inclusion dependency $R_p[x_1, \ldots, x_m] \subseteq R_c[x_1, \ldots, x_m]$.

The inverse of **C2**, $R_c[x_1, \ldots, x_m] \subseteq R_p[x_1, \ldots, x_m]$, always holds, because RXL's semantics always define a tree. Therefore, **C2** implies $\pi_{x_1,\ldots,x_m}(R_c) = R_p$ in this context. In general, the problem of checking whether a given set of functional and inclusion dependencies implies another set of dependencies is undecidable [1]. SilkRoute uses heuristics and known algorithms for restricted problems. In particular, it does not consider inclusion dependencies when it checks if a functional dependency can be derived, which allows the check to be done in linear time [2]. From our experience, this solution is adequate for typical RXL queries.

In the second step, the view tree's nodes are grouped into equivalence classes: each class contains nodes that are reachable only by '1'-labeled edges. Fig. 11 illustrates this step. For each such class, a new Skolem term S is created, and a new datalog rule $S(v_1, \ldots, v_m)$:- Q is created, where, $v_1 \ldots v_m$, are the union of the Skolem-term arguments of each node in the class, and Q' is the

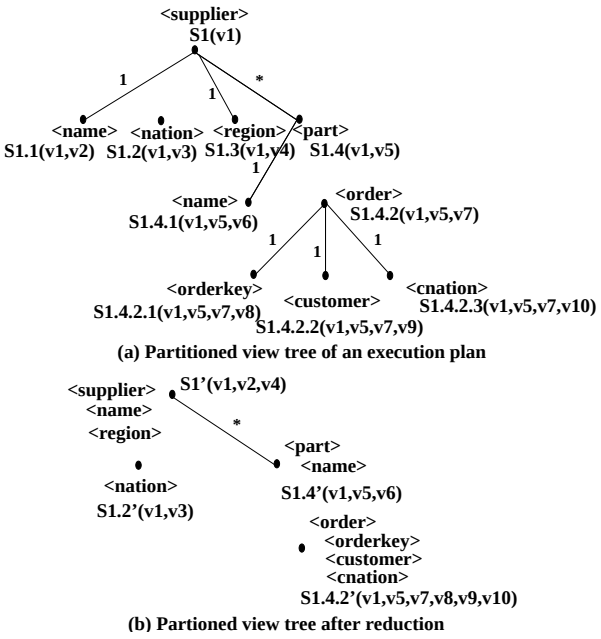


Figure 11: Example of view tree reduction

conjunction of all the nodes' query bodies. In each class, the greatest-common-ancestor node is replaced by the node $S(v_1, \ldots, v_m)$. In Fig. 11, the equivalence classes are {`S1, S1.1, S1.3`}, {`S1.4, S1.4.1`}, and {`S1.4.2, S1.4.2.1, S1.4.2.2, S1.4.2.3`}. They are replaced by `S1'`, `S1.4'`, and `S1.4.2'`, respectively.

Two potential benefits of view-tree reduction are that it can reduce the number of outer joins[2], and it can reduce the total size of the relations and therefore, the total size of data transferred. In general, whether view-tree reduction actually decreases the data size depends on the characteristics of submitted queries and database instances. For example, in Fig. 11, if the data size of `<region>` element dominates, then in the reduced view tree, its large data value would occur in every tuple in the relation for `S1'`, which could increase data-transfer time. Both query-only time and data-transfer time of a reduced plan, therefore, may not always be faster than the corresponding non-reduced plan. To alleviate this problem, we can prohibit the reduction of specific nodes based on the average data size estimated by the target database. We use view-tree reduction as a plan-improving heuristic: given a set of arbitrary non-reduced plans, the corresponding set of reduced plans, in general, are more efficient. Our experimental results support this heuristic.

4. EXPERIMENTS

One important feature of a view tree is that it permits us to generate and compare all possible execution plans for an RXL query. As discussed in Sec. 2, other XML

[2]The current query generator constructs for each node an outer join with the union of its children, which disappears when all children are labeled '1'.

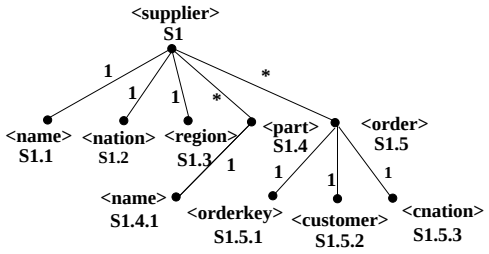


Figure 12: Labeled view tree for Query 2

Config	Size	Database Server Platform	Client Platform
A	1 MB	AMD K6-2 350 MHz 256MB mem 1GB swap Linux RH 6.1	SGI Challenge L 4GB mem IRIX64 V6.5
B	100 MB	Intel Celeron 566 MHz 256 MB mem 1GB swap Linux RH 6.2	Intel Pentium III 192 MB mem Linux RH 6.2

Table 1: Experimental Configurations

publishing systems produce either a unified or fully partitioned plan. Here, we present experiments that compare these default plans to the "optimal" plans, i.e., those plans that have the fastest execution times compared to all others. SilkRoute uses an outer-join strategy to generate plans, so its unified plans are not equivalent to outer-union plans [9]. For completeness, we include a unified outer-union plan in the experiments. We also compare plans generated from non-reduced view trees with those generated from reduced view trees.

We also generated plans for Query 2. Query 2 is identical to Query 1 except that the block defining the `order` node is a child of the `supplier` node instead of the `part` node. Fig. 12 depicts its view tree. In Query 1, the two one-to-many edges (labeled "*"), are nested in a chain, whereas in Query 2, the two '*' edges are parallel. A '*' edge corresponds to a outer join in an SQL query, so each query stresses an SQL engine in a different way: Query 1 has nested outer joins and Query 2 has unions of outer joins.

The experiments were run using two database configurations (Table 1). Configuration A used the TPC-H Database with 1 MB of data, and Configuration B used a 100 MB database. Exhaustive query plans were generated for Config. A; Config. B is used in Sec. 5 to evaluate our plan-generation algorithm. Due to licensing restrictions, we are not permitted to identify the commercial product used in our experiments. In the experiments, the database client was a simple Java program that submitted SQL queries to the database server and read tuples from the tuple streams via JDBC. Both configurations use JDK 1.2 and JDBC 1.2.2.

Recall that each view tree has ten edges. As described in Sec. 3, one plan is generated for each subset of edges in the view tree, so there are 512 possible plans for each query. Each plan generates between one and ten SQL queries, each of which produces one tuple stream.

Figures 13 and 14 plot the execution times of the 512 plans for Queries 1 and 2, respectively. The horizontal axis is the number of tuple streams per plan and the vertical axis is the execution time in milliseconds, on a log scale. Both total time and query-only time were measured on the SilkRoute client. Total time includes query execution time on the database server and data transfer time to the client: timing began when the first SQL query was submitted to the server and terminated when the *last* tuple was read from the last tuple stream. Query-only time includes the time until the *first* tuple is read from a tuple stream. The time to first tuple is comparable to the time to count all tuples in the result on the server only, so pipelining of output during query execution did not effect our measurements. If a subquery did not complete within 5 minutes, no time was reported. For Query 1, 101 plans timed out; for Query 2, no plans timed out.

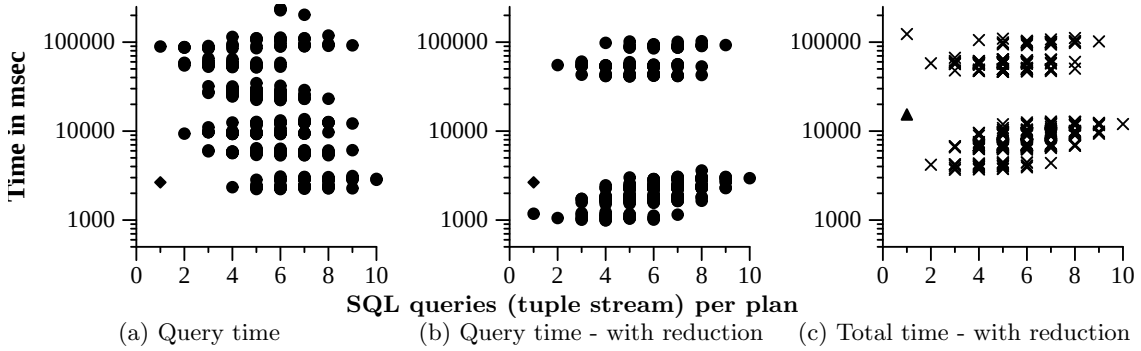
For non-reduced trees, the outer-union and fully partitioned plans are slightly slower than the optimal plans. Figures 13(a) and 14(a) plot the query-only time for non-reduced trees. In Fig. 13(a), the unified outer-union plan (diamond) is 16% slower than optimal and the fully partitioned plan is 24% slower. In Fig. 14(a), the outer-union plan is 21% slower and the fully partitioned plan is 41% slower.

Recall from Sec. 3.5 that view-tree reduction eliminates redundant queries in a view tree. To determine the effect of view-tree reduction on execution time, we generated 512 plans for Queries 1 and 2 and then applied the view-tree reduction algorithm to each plan. Figs. 13(b) and 14(b) contain the query-only times of the plans with view-tree reduction. These graphs should be compared to Figs. 13(a) and 14(a), respectively. Note that view-tree reduction significantly reduces query-only time. For both Queries 1 and 2, the ten fastest reduced plans are 2.5 times faster than the ten fastest non-reduced plans, and the optimal plans are 2.6 to 4.3 times faster than the outer-union and fully partitioned plans.

The differences for total execution times, which include data-transfer time, are similar. For Query 1 in Fig. 13(c), the unified outer-union (triangle) is four times slower than optimal, and the fully partitioned plan is three times slower. For Query 2 in Fig. 14(c), the unified outer-union plan is 4.8 slower than optimal, and the fully partitioned plan is 3.7 times slower.

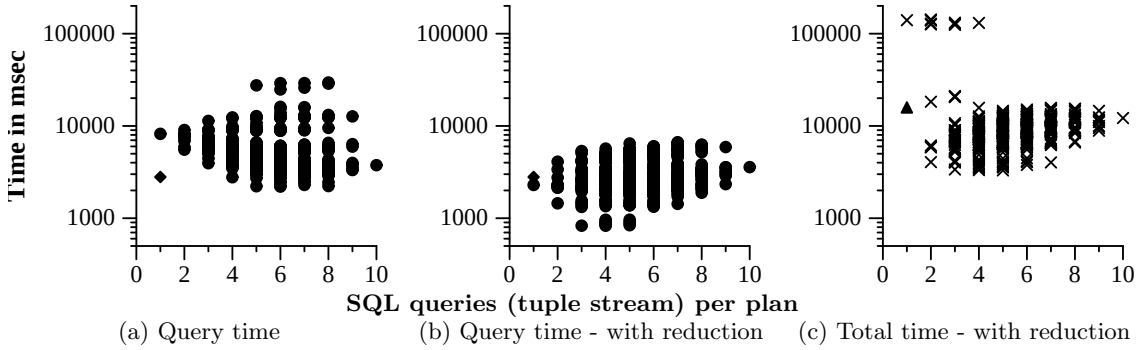
We note that for query-only time, the unified outer-union plan is only slightly slower than the unified outer-join plan, but its total execution time is much faster. The outer-join plan actually produces fewer, but wider, tuples than the outer-union plan; the additional width may induce anomalous caching behavior in JDBC. This suggest that we could further improve the total running time of the best plans if we rewrite them from outer joins to outer unions.

5. PLAN-GENERATION ALGORITHM



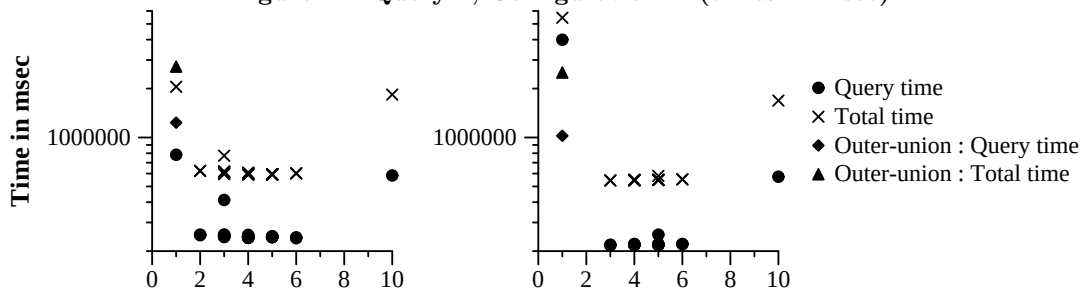
(a) Query time (b) Query time - with reduction (c) Total time - with reduction

Figure 13: Query 1, Configuration A (times in msec)



(a) Query time (b) Query time - with reduction (c) Total time - with reduction

Figure 14: Query 2, Configuration A (times in msec)



(a) Query 1 (b) Query 2

Figure 15: Configuration B, with view-tree reduction (times in msec)

The experiments indicate that choosing a default unified, fully partitioned, or purely heuristic execution plan is not effective in practice and that devising an algorithm to generate near-optimal plans is worthwhile. The graphs in Fig. 13 and 14 also suggest that there are many near-optimal queries to choose from. The only reliable source of query costs is the target RDBMs. Given that the target RDBMs can estimate the cost of a query, we can use the target database to choose "good" edges in a view tree, i.e., those edges whose two associated queries are less expensive to evaluate together than separately.

Here, we present an algorithm that given a view tree, returns an evaluation plan that contains a set of mandatory view-tree edges and a set of optional view-tree edges. The algorithm uses the RDBMs to estimate the relative cost of an edge in the view tree. For an edge $e = (sfi_1, sfi_2)$, where sfi_1, sfi_2 are the Skolem functions associated with the edge's parent and child nodes respectively, we compare the sum of the costs of evaluating the queries associated with sfi_1 and sfi_2 to the cost of evaluating the two queries combined. We use a simple

linear equation to estimate a query's cost:

$$\text{cost}(q, a, b) = a * \text{evaluation\_cost}(q) + b * \text{data\_size}(q)$$

$$\text{data\_size} = f(|\text{attrs}(q)| * \text{cardinality}(q))$$

The coefficients a and b assign weights to the query evaluation cost and query data size, respectively. The RDBMs serves as an oracle, providing the values for the functions `evaluation_cost` and `cardinality`. This technique is feasible, because most commercial databases provide support for estimating these costs.

Fig. 17 contains the plan-generation algorithm **genPlan**. Fig. 16 contains the type signatures for the algorithm's functions. The function `genPlan` takes a view tree *ViewTree*, the cost coefficients a and b described above, and two thresholds: t_1 is the maximum threshold for a mandatory edge and t_2 is the maximum threshold for an optional edge. The recursive function `addEdge` takes the current set of edges (*Edges*), the queries associated with those edges (*Queries*), and the current sets of mandatory and optional edges. On each recursive invocation, `addEdge` computes the relative costs of every edge e_i in *Edges*:

$$cost = \texttt{cost}(q_c) - (\texttt{cost}(q_1) + \texttt{cost}(q_2))$$

where q_1 and q_2 are the queries associated with e_i's parent and child nodes, and q_c is the result of combining q_1 and q_2. These costs are then sorted and `addPlan` considers the edge e with smallest relative cost (i.e., the one with greatest combined benefit). If the relative cost of e is less than t_1, the maximum threshold of a mandatory edge, then e is added greedily to the mandatory edges of the plan. Similarly, if e's relative cost is less than t_2, it is added to the optional edges of the plan. The function `addEdge` greedily adds edges until no remaining edge is less than the mandatory or optional threshold.

The function `combineQueries` determines how to collapse two queries into one query based on the label of the edge in the view tree. The '1'-labeled edges correspond to inner joins and '*'-labeled edges to outer joins. In addition, `combineQueries` applies view-tree reduction to eligible edges.

The complexity of the function `genPlan` is $O(|Edges|^2)$, because `addEdge` recomputes the costs of every edge in the view tree on each recursive call. This is not necessary : instead, it could recompute the costs of those edges incident to each edge e selected by `addEdge`, but to simplify presentation of the algorithm, this definition recomputes all the edge costs on each invocation.

```
function genPlan(ViewTree, t1, t2, a, b) {
  function addEdge(Edges, Queries, mandE, optE) {
    // Compute relative cost of each edge in Edges
    costE :  {Cost} = ⋃
      for e_i in Edges {
        let (sfi_1, sfi_2) = e_i
                q_1 = getQuery(sfi_1, Queries)
                q_2 = getQuery(sfi_2, Queries)
                q_c = combineQueries(q_1, q_2, e_i)
        in (cost(q_c) - (cost(q_1) + cost(q_2)), e_i, q_c)
      }
    // Sort edges by costs
    sortedE = sort costE
    // Greedily add "best" edge to plan
    (i, e, q_c) = head(sortedE)
    if (i < t1 || i < t2) {
      let (sfi_q, svt_q, body_q) = q_c
        // Add e to plan
        mandE' = if (i < t1) mandE ∪ {e} else mandE
        optE' = if (i >= t1 && i < t2) optE ∪ {e}
                else optE
        (sfi_1, sfi_2) = e
        // Remove edge e from Edges
        Edges' = Edges - {e}
        // Remove e's queries from Queries
        Queries' = (Queries - { getQuery(sfi_1, Queries) }) -
                { getQuery(sfi_2, Queries) }
        // Add combined query q_c to Queries
        Queries'' = Queries' ∪ {q_c}
        // Remove edges incident to e from Edges
        incidentE = incidentEdge(Edges, e)
        Edges'' = Edges' - incidentE
        // For each edge incident to e, add new edge
        // that is incident to combined node defined
        // by query q_c
        Edges''' = Edges'' ⋃
                for i in incidentE {
                    let (sfi_u, sfi_v) = i in
                    if (sfi_u == sfi_1 || sfi_u == sfi_2)
                    { (sfi_q, sfi_v) }
                    else { (sfi_u, sfi_q) }
                }

      in addEdge(Edges''', Queries'', mandE', optE')
    } else (mandE, optE)
  }
  let (Edges, Queries) = ViewTree
  in addEdge (Edges, Queries, {}, {})
}
```

Figure 17: Greedy algorithm for plan generation

5.1 Results

We applied the plan-generation algorithm twice to the view trees for Query 1 and Query 2: in one case, `combineQueries` did not apply view-tree reduction and in the second, it did. The generated plans for Query 1 appear in Fig. 18 (a) and (b), and in Fig. 18 (c) and (d) for Query 2.

The most important result is that the generated plans correspond directly to the fastest plans measured in Sec. 4. For Query 1, the plans generated from the non-reduced and reduced view trees correspond to the fastest 32 plans. For Query 2, the plans generated from the non-reduced view tree correspond to the fastest 32 plans, and the plans generated from the reduced view tree correspond to the first 31 and the 34th fastest plans. In Configuration B, the size of the database was 100 MB, so it was not possible to exhaustively test all 512 plans. Instead, we ran the greedy algorithm using view-

Edge = **SFI** × **SFI**	A view-tree edge is a pair of Skolem-function indices
Query = **SFI** × **SVI** × **Body**	A query is a Skolem-term and a body of relation names and filters
ViewTree = {**Edge**} × {**Query**}	A view tree contains a set of edges and a set of queries
Cost = **Int** × **Edge** × **Query**	The cost of an edge, the edge, and the query if the edge is collapsed
`getQuery` : **SFI** × {**Query**} → **Query**	
`incidentEdge` : E : {**Edge**} × e : **Edge** → [**Edge**]	Returns edges in E incident to e
`combineQueries` : **Query** × **Query** × **Edge** → **Query**	Combines two queries on given edge into one query
`addEdge` : E : **Edge** × *plan* : **Edge**	Sorts edges in E by costs and adds qualifying edge to *plan*
`genPlan` : **ViewTree** × **Int** × **Int** × **Int** × **Int** → {**Edge**} × {**Edge**}	Returns plan containing mandatory and optional edges

Figure 16: Types and functions of greedy algorithm

tree reduction and compared the generated plans with the unified and fully partitioned plans. Sixteen plans were generated for Query 1; they appear in Fig. 18 (b). (Each subset of the four optional edges defines a plan.) Eight plans were generated for Query 2; they appear in Fig. 18 (d). Fig. 15 plots the query-only and total-execution times for these plans and for the unified outer-union and fully partitioned plans.

For Query 1 in Fig. 15(a), the query-only time of the outer-union was five times slower than the optimal plan and the fully partitioned plan was 2.4 times slower. For Query 2, the differences were similar; the outer-union plan was 4.7 times slower than the optimal plan and the fully partitioned plan was 2.6 times slower. These results indicate that as the size of the XML view increases, generating optimal plans becomes imperative. Comparing total execution times for both Query 1 and 2, the outer-union plan was 4.6 times slower and the fully partitioned plan was 3.1 times slower.

For all the plans generated, we used the same values for the coefficients a (100) and b (1) and the thresholds t_1 (-60000) and t_2 (6000), which indicates that the linear cost function depends primarily on the characteristics of the database environment, and not on the characteristics of the query. Further experiments using a larger set of test queries are necessary to confirm this hypothesis.

Recall that the complexity of the plan-generation algorithm is $O(|Edges|^2)$ and that on *each* edge access, the algorithm requests the *estimated* costs of evaluation time and data size from the target database's query optimizer. For Queries 1 and 2, we found that the actual number of database requests for query-cost estimates were much smaller than the expected number of requests ($9^2 = 81$). Both Queries 1 and 2 required 22 requests for the non-reduced view tree and 25 requests for the reduced view tree.

6. RELATED RESEARCH AND SYSTEMS

Shanmugasudaram et al. [9] describe several methods for computing XML views with relational engines. They classify the methods along three axis: early/late structuring, early/late tagging, and in-engine/outside-engine XML generation. They consider a variety of algorithms and compare them experimentally. In the *unordered outer union* strategy, the tagger uses a main memory hash table to assemble the XML objects, which requires the XML view fit in main memory. In *CLOB De-correlated queries*, the XML result is constructed by the relational engine, which is also effective when the XML view fits in main memory. The best overall performance is achieved by the CLOB de-correlated algorithm, the unsorted outer union, and the sorted outer union. Of these, only the sorted outer union applies to large XML views that exceed main memory. In Sec. 7, we discuss when the outer-union plan is likely to perform as well as the optimal plans found by SilkRoute.

Three commercial XML publishing systems: Oracle XML SQL Utility [10], IBM DB2 XML Extender [12], and Microsoft SQL Server 2000 [6], support features similar to those provided by SilkRoute. All three provide languages for defining XML views. Oracle's XSQL embeds individual SQL queries in XSLT [13] stylesheets. The result of the SQL query is emitted in a canonical XML format, and the stylesheet converts the XML into the desired view. Of these systems, XSQL couples most tightly the XML view to the corresponding SQL queries. The IBM DB2 Data Access Definition (DAD) language, like RXL, has a data extraction part and an XML template. Each element in the XML template may contain arbitrary selection and join conditions on the relational tables. Unlike RXL, the criteria for grouping elements is implicit in the DAD, and DAD specifications cannot be nested arbitrarily. SQL Server 2000 supports XML view mechanisms like the two described above. In addition, the user may construct the unified SQL plan by hand. This effectively hard wires the evaluation plan into the view, but it allows the user to define arbitrarily complex XML views. Hand-written unified queries are similar to those constructed automatically by SilkRoute's plan-generation algorithm.

Although RXL is specific to SilkRoute, it can express the transformations provided by the three XML publishing tools described above. SilkRoute's view tree representation of XML view queries captures the XML mappings in all these systems. Our greedy optimization algorithm takes a view tree as input, and therefore could be directly applied to the XML view definitions expressed by these tools.

7. DISCUSSION

Generating SQL queries from an XML view definition is a tedious task, and as we have shown, different SQL-generation strategies dramatically effect query-evaluation time. These observations indicate that the user of a relational-to-XML publishing system should not be responsible for choosing SQL queries. To bet-

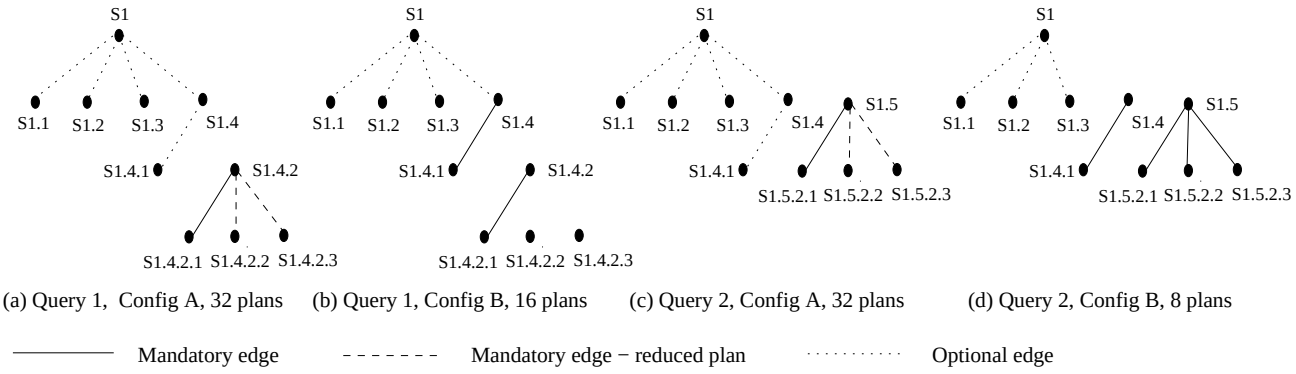


Figure 18: Plans selected by Greedy Algorithm

ter support large XML views, we presented a method that decomposes the XML view definition into several, smaller SQL queries and submits the decomposed SQL queries to the target database. Our greedy algorithm for decomposing an XML view definition relies on query-cost estimates from the target query optimizer. This method works well in practice and generates execution plans that are near optimal. Although particularly effective in an XML middle-ware system, our view-tree representation can encompass the view-definition languages of commercial relational-to-XML systems. Commercial systems typically generate XML in-engine, because the cost of binding application variables to the tuples dominates execution time [9]. Our decomposition method could be applied within a relational query optimizer as a preprocessing step to XML publishing of relational data in-engine.

This work is focussed on publishing large XML documents in an environment in which the middle-ware system has no control over the physical environment or query optimizer of the target database. Given these constraints, our greedy algorithm for searching for optimal query plans is necessary and effective. The simpler outer-union strategy, however, might be adequate when the middle-ware system has more control over the target database. SilkRoute's generated optimal plans do better than the unified outer-union plan, because each individual query is smaller than the outer-union plan. Small queries are less likely to stress the query optimizer; they sort smaller result relations and therefore are less likely to spill tuples to disk; and they typically have many fewer null values than a unified query. An outer-union plan can be reduced by hand, which would provide the same benefits as automatic view-tree reduction. Assuming that the target database has plentiful memory and/or multiple disks, and efficiently supports null values, the resulting outer-union plan is likely to be comparable to SilkRoute's generated optimal plans. Finally, the outer-union plan may also be appropriate when a user query requests only a subset of the XML view, and the result document is small. In this scenario, the outer-union strategy should work well, because the resulting SQL query is usually simple. This scenario is considered in [5], where the XML view of the database is virtual, and users query it using XML-QL.

Acknowledgements. Many thanks to Jai Shanmugasudaram for his detailed and insightful comments.

8. REFERENCES

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

[2] C. Beeri, P Bernstein, "Computational Problems Related to the Design of Normal Form Relational Schemes", ACM Transactions on Database Systems, V4, #1, pp 30–59, 1979

[3] M. Carey, et al, "XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents", VLDB 2000, pp 646-648.

[4] A. Deutsch, M. Fernández, D. Florescu, A. Levy, and D. Suciu. "A Query Language for XML", WWW8, 1999.

[5] M. Fernández, W Tan, D Suciu, "SilkRoute : Trading between Relations and XML", WWW9, 2000.

[6] M. Rys, Microsoft, Support WebCast: Microsoft SQL Server 2000: New XML Features, April, 2000 (http://support.microsoft.com/servicedesks/ `Webcasts/wc042800/wcblurb042800.asp`)

[7] A. Sahuguet, "Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask", SIGMOD WebDB Workshop 2000, pp 69-74.

[8] P. Selinger, et al, "Access Path Selection in a Relational Database Management System", SIGMOD, pp 23-34, 1979.

[9] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, B. Reinwald, "Efficiently Publishing Relational Data as XML Documents", VLDB 2000, pp 65-76.

[10] B. Wait, Oracle Corporation, "Using XML in Oracle Database Applications", Nov., 1999, (http://technet.oracle.com/tech/xml/info /htdocs/otnwp/about_xml.htm)

[11] Transaction Processing Performance Council, TPC-H (ad-hoc, decision support) benchmark, http://www.tpc.org/

[12] XML Extender Administration and Programming, "IBM DB2 Universal Database XML Extender", (`http://www-4.ibm.com/software/data/db2/` extenders/xmlext/docs/v71wrk/english/index.htm)

[13] World-Wide Web Consortium XSL Transformations (XSLT), Version 1.0. W3C Recommendation, Nov., 1999. `http://www.w3.org/TR/xslt/`.