Teaching Relational Optimizers About XML Processing

Sihem Amer-Yahia, Yannis Kotidis, and Divesh Srivastava

AT&T Labs-Research, Florham Park NJ 07932, USA,
{sihem,kotidis,divesh}@research.att.com

Abstract. Due to their numerous benefits, relational systems play a major role in storing XML documents. XML also benefits relational systems by providing a means to publish legacy relational data. Consequently, a large volume of XML data is stored in and produced from relations. However, relational systems are not well-tuned to produce XML data efficiently. This is mainly due to the flat nature of relational data as opposed to the tree structure of XML documents. In this paper, we argue that relational query optimizers need to incorporate new optimization techniques that are better suited for XML. In particular, we explore new optimization techniques that enable computation sharing between queries that construct sibling elements in the XML tree. Such queries often have large common join expressions that can be shared through appropriate rewritings. We show experimentally that these rewritings are fundamental when building XML documents from relations.

1 Introduction

Relational systems are good for XML and XML is good for relations. On the one hand, there are mature relational systems that can be used to store the ever growing number of XML documents that are being created. On the other hand, XML is a great interface to publish and exchange legacy relational data. Consequently, most XML data today comes from and ends up in relations. Unfortunately, XML and relational data differ in their very nature. One is tree-structured, the other is flat and often normalized. Consequently, the performance of relational engines varies considerably when producing XML documents from relations. In research, several efforts explored how to build XML documents from relations efficiently [2, 4, 5, 7, 9]. In particular, the authors in [7] argue for extending relational engines to benefit XML queries. In this work, we explore complementary *optimization* techniques that are fundamental to handle XML queries efficiently in a relational engine.

Due to the flat nature of relational data, as opposed to the nested structure of XML, generating an XML document from relations often involves evaluating multiple SQL queries (possibly as many as the number of nodes in the DTD). These queries often contain common sub-expressions in order to build the tree structure. Thus, query performance can vary considerably, necessitating a cost-based optimization of the plan for building XML documents. In [9], the authors explore rewriting-based optimizations between a query for a parent node and the queries for its children nodes in a middle-ware environment. We argue that sharing computation between queries for sibling nodes, not just between parent and children queries, is key to efficiently building XML documents from relational data, both in a middle-ware environment (as in [9]) and for the query optimizer of a relational system.

Consider a publishing example where we want to build XML documents from a TPC-H database [16]. These documents conform to a DTD with `Customer` as a root element and its two sub-elements `SuppName` (for suppliers) and `PartName` (for parts). In order to identify the suppliers of a given customer, the TPC-H table, `CUSTOMER`, is joined with `ORDERS` and `LINEITEM`. This join expression needs to be joined with the `SUPPLIER` table to compute supplier names (i.e., element `SuppName`). This same join expression (between `CUSTOMER`, `ORDERS` and `LINEITEM`) needs to be joined with the `PART` table to evaluate the set of part names associated with each customer (i.e., `PartName`). Obviously, the sibling queries at the two elements `SuppName` and `PartName` share a large common join expression and could be merged into a single query (using an outer union) where this common join expression is factored out, enabling the relational engine to evaluate it only once. In [9], every such query merge has to go through a parent/child merge. For example, the two sibling queries at `SuppName` and `PartName` can be merged only if they are also joined with the query at `Customer`, resulting in a single query that is used to evaluate the whole document. If `Customer` is a "fat" node (containing multiple attributes such as `Name`, `Zip` and `Phone`), the entire customer information will be replicated with each `SuppName` and each `PartName` (because of outer-joins) which may result in higher communication costs (in a middle-ware environment) and computation costs (both in a middle-ware environment, and in a relational optimizer). Being able to merge sibling queries independently from their parent query and factor out common sub-expressions in merged queries is a key rewriting that we will explore when building XML documents from relations.

Our contributions are as follows:

– We show that sharing computation between sibling queries is a fundamental optimization for efficient building of XML documents from flat relational data.
– We describe several query rewritings that exploit shared computation between queries used to build an XML document.
– We design an optimization algorithm that applies our rewritings to find the best set of SQL queries that optimizes processing time and achieves a good compromise between processing and communication times in a middle-ware environment.
– We run experiments that compare multiple strategies of sharing common computation between sibling queries and identify their considerable performance benefits.

Section 2 describes our motivating examples and gives a formal definition of our problem. Rewriting techniques are presented in Section 3. Section 4 contains the optimization algorithm and a study of the search space. Experimental results are presented in Section 5. Related work is discussed in Section 6. Section 7 concludes.

2 Motivation and Problem Definition

2.1 Publishing Legacy Data in XML

We consider a simplified version of the relational schema of the TPC-H benchmark [16]. This schema describes parts ordered by customers and provided by suppliers.

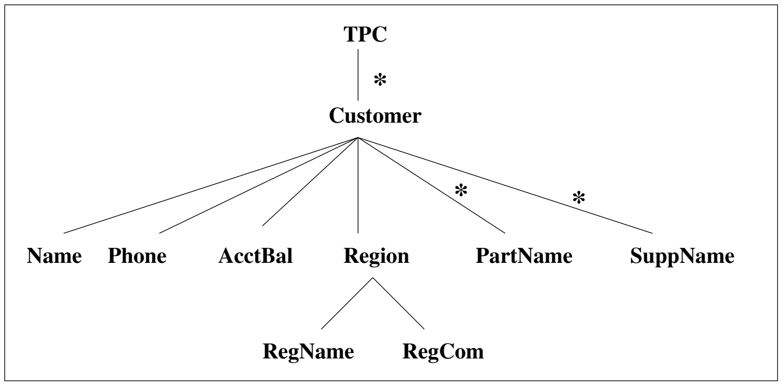

Fig. 1. Publishing Legacy Data: Example DTD

```
CUSTOMER[C_CUSTKEY,C_NAME,C_PHONE,C_ACCTBAL,C_NATIONKEY]
NATION[N_NATIONKEY,N_NAME,N_REGIONKEY]
REGION[R_REGIONKEY,R_NAME,R_COMMENT]
PART[P_PARTKEY,P_NAME]
SUPPLIER[S_SUPPKEY,S_NAME,S_NATIONKEY]
ORDERS[O_ORDERKEY,O_CUSTKEY]
LINEITEM[L_ORDERKEY,L_PARTKEY,L_SUPPKEY]
```

We want to build XML documents that conform to the DTD given in Fig. 1. To keep the exposition simple, we represent this DTD as a tree. Edges labeled with a '*' are used for repeated sub-elements. We are interested only in publishing information about customers whose account balance is lower than `$5000` (predicate p). `PartName` contains the names of the parts ordered by a customer. `SuppName` contains the names of their suppliers.

It has been shown previously in [5] that it is possible to write a single SQL query (containing outer-joins and possibly outer-unions) to build an XML document for a relational database regardless of the underlying relational schema. It has also been shown in [9] that a set of SQL queries that are equivalent to the single query can be generated. In order to better explain performance issues, we examine the case where an SQL query is generated for each element in the DTD as follows (where $C = \sigma_p$ (CUSTOMER), $J1 = C \bowtie_{\texttt{NATIONKEY}}$ NATION $\bowtie_{\texttt{REGIONKEY}}$ REGION and $J2 = C \bowtie_{\texttt{CUSTKEY}}$ ORDERS $\bowtie_{\texttt{ORDERKEY}}$ LINEITEM):

$Q_{\texttt{Customer}} = \pi_{\texttt{C\_CUSTKEY}}(C)$
$Q_{\texttt{Name}} = \pi_{\texttt{C\_CUSTKEY,C\_NAME}}(C)$
$Q_{\texttt{Phone}} = \pi_{\texttt{C\_CUSTKEY,C\_PHONE}}(C)$
$Q_{\texttt{AcctBal}} = \pi_{\texttt{C\_CUSTKEY,C\_ACCTBAL}}(C)$
$Q_{\texttt{Region}} = \pi_{\texttt{C\_CUSTKEY,R\_REGIONKEY}}(J1)$
$Q_{\texttt{RegName}} = \pi_{\texttt{C\_CUSTKEY,R\_REGIONKEY,R\_NAME}}(J1)$
$Q_{\texttt{RegCom}} = \pi_{\texttt{C\_CUSTKEY,R\_REGIONKEY,R\_COMMENT}}(J1)$
$Q_{\texttt{PartName}} = \pi_{\texttt{C\_CUSTKEY,P\_NAME}}(J2 \bowtie_{\texttt{PARTKEY}} \texttt{PART})$

$$Q_{\texttt{SuppName}} = \pi_{\texttt{C\_CUSTKEY},\texttt{S\_NAME}}(J2 \bowtie_{\texttt{SUPPKEY}} \texttt{SUPPLIER})$$

Several sibling queries share common expressions. The simplest example is the case of $Q_{\texttt{Name}}$, $Q_{\texttt{Phone}}$ and $Q_{\texttt{AcctBal}}$ that are all projections on the same (subset of the) CUSTOMER table. This is due to the fact that some fields in this table are used to generate sub-elements. Thus, these sibling queries could be merged into a single query Q_C. The merged query could further be merged with the parent query, $Q_{\texttt{Customer}}$, resulting in "fat" customer nodes as follows:

$$Q_C = \pi_{\texttt{C\_CUSTKEY},\texttt{C\_NAME},\texttt{C\_PHONE},\texttt{C\_ACCTBAL}}(C)$$

The second example involves $Q_{\texttt{RegName}}$ and $Q_{\texttt{RegCom}}$ that share a common join expression $J1$. This is due to the fact that nations and regions are normalized into tables and that recovering them requires performing joins with these intermediate tables. By merging $Q_{\texttt{RegName}}$ and $Q_{\texttt{RegCom}}$ into a single query, the common expression is evaluated only once:

$$Q_{\texttt{RegNameCom}} = \pi_{\texttt{C\_CUSTKEY},\texttt{R\_REGIONKEY},\texttt{R\_NAME},\texttt{R\_COMMENT}}(J1)$$

Furthermore, $Q_{\texttt{RegNameCom}}$ could be merged with Q_C resulting in:

$$\pi_{\texttt{C\_CUSTKEY},\texttt{C\_NAME},\texttt{C\_PHONE},\texttt{C\_ACCTBAL},\texttt{R\_REGIONKEY},\texttt{R\_NAME},\texttt{R\_COMMENT}}(J1)$$

The last and most interesting example is the case of the two sibling queries $Q_{\texttt{PartName}}$ and $Q_{\texttt{SuppName}}$ that share a common join expression $J2$. In order to share this join expression, the two queries could be merged. However, due to the fact that a customer has multiple parts and suppliers, merging $Q_{\texttt{PartName}}$ and $Q_{\texttt{SuppName}}$ might result in replication. Replication might slow down query processing as well as communication time. There are two ways to avoid replication in this case. Either the relational optimizer is able to optimize outer unions and the queries are rewritten using an outer-union where the common sub-expression is factored out. Or, the relational engine is forced to compute $J2$, materialize it and then use it to evaluate the two queries. The query $Q_{\texttt{PartSupp}}$ that results from the outer union is given by:

$$(\pi_{\texttt{C\_CUSTKEY},\texttt{P\_NAME},\texttt{NULL}}(J2 \bowtie_{\texttt{PARTKEY}} \texttt{PART})) \cup (\pi_{\texttt{C\_CUSTKEY},\texttt{NULL},\texttt{S\_NAME}}(J2 \bowtie_{\texttt{SUPPKEY}} \texttt{SUPPLIER}))$$

Because of the presence/absence of some indices, it might not always be the case that the merged query, $Q_{\texttt{PartSupp}}$ is cheaper than the sum of $Q_{\texttt{PartName}}$ and $Q_{\texttt{SuppName}}$. The relational optimizer could choose different plans to evaluate the common join expression in the two queries resulting in a better evaluation time than the merged query.

Finally, using an outer-join, it is possible to rewrite all of the nine queries above into a single query. That outer-join guarantees that all customers satisfying predicate p will be selected (even if they have never ordered any part). However, if all queries are merged, each customer tuple (along with its required fields) will be replicated as many times as the number of parts and suppliers for this customer. This replication is due to merging queries with their parent query and impacts computation time as well as query result size. Therefore, depending on the amount of replication generated by merging

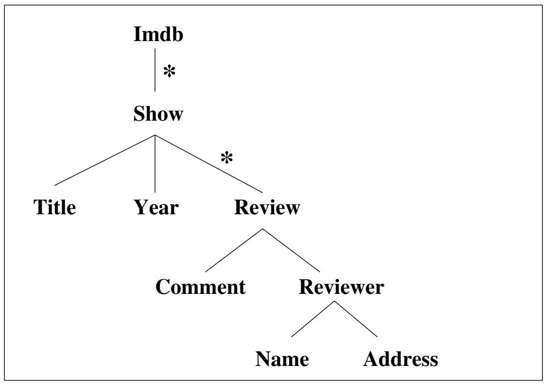

Fig. 2. Shredded XML: Example DTD

sibling queries with their parent query, it might be desirable to optimize queries at siblings *separately* from their parent query.

A key observation is that shared computation between sibling queries, when building XML data from relations, is often higher than shared computation between parent and children queries making sibling merges more appropriate than merging with the parent query. This is particularly true in two cases: (i) if the relational schema is highly normalized and thus, several joins involving intermediate relations are needed to compute sub-elements, and (ii) if the DTD contains repeated sub-elements that might create replication (of the parent node) when children and parent queries are merged together.

2.2 Building XML Documents from Shredded XML

There have been many efforts to explore different shredding schemas for XML into relations [1]. Most of them rely on using key/foreign key relationships to capture document structure and thus, generate sibling queries with common sub-expressions when building XML documents.

Fig. 2 contains the DTD of a document that has been shredded and stored in a relational database. The example contains information on shows such as their title and year as well as the reviews written by a reviewer who has a name and an address. We first consider the relational schema `Schema1` given below:

```
SHOW[SHOW_ID,TITLE,YEAR]
REVIEW[REVIEW_ID,COMMENT,SHOW_ID]
REVIEWER[REVIEWER_ID,NAME,ADDRESS_ID,REVIEW_ID]
ADDRESS[ADDRESS_ID,ADDRESS]
```

In order to compute the `Name` and `Address` of each reviewer in a review associated with a show, the following two queries are needed (where $J = \texttt{REVIEW} \bowtie_{\texttt{REVIEW\_ID}} \texttt{REVIEWER}$):

$$Q_{\texttt{Name}} = \pi_{\texttt{SHOW\_ID},\texttt{REVIEW\_ID},\texttt{REVIEWER\_ID},\texttt{NAME}}(J)$$
$$Q_{\texttt{Address}} = \pi_{\texttt{SHOW\_ID},\texttt{REVIEW\_ID},\texttt{REVIEWER\_ID},\texttt{ADDRESS}}(J \bowtie_{\texttt{ADDRESS\_ID}} \texttt{ADDRESS})$$

These queries share a common sub-expression J. If the address of a reviewer was inlined inside a review in the relational schema, no redundant computation would have occurred. This is illustrated in `Schema2` as follows:

```
SHOW[SHOW_ID,TITLE,YEAR]
REVIEW[REVIEW_ID,COMMENT,NAME,ADDRESS,SHOW_ID]
```

In this case, $Q_{\texttt{Name}}$ and $Q_{\texttt{Address}}$ correspond to simple projections on the `REVIEW` table and could be easily merged into a single query.

Even when a query workload is used for XML storage in relations, as in [3], our optimization techniques for SQL "publishing" queries are still useful for two reasons. First, an improved optimizer can provide more accurate cost estimates for the queries in the workload, for making cost-based shredding decisions. Second, queries that are not in the initial query workload would still need to be optimized in an ad-hoc fashion.

2.3 Problem Definition

We are interested in building XML documents from a relational store efficiently. Following the approaches used in [4, 9], the structure of the resulting XML documents is abstracted as a DTD-like labeled tree structure, with element tags serving as node labels, and edge labels indicating the multiplicity of a child element under a parent element; it is important to note that multiple nodes in this DTD-like structure may have the same element tag, due to element sharing, or due to data recursion.

SQL queries are associated with nodes in this DTD-like structure, and together determine the structure and content of the resulting XML documents that are built from the relational store; the examples in Section 2.1 are illustrative. The set of SQL queries generated for a given XML document might be small. However, the amount of data that these queries manipulate can be large and thus, optimization is necessary. If we denote by $\mathcal{S}$ the set of all SQL queries necessary to build a document conforming to a given DTD, then the problem is formulated as follows:

find the set of SQL queries $\mathcal{S}$ *such that* $\Sigma_{s\in\mathcal{S}}(w_p * proc(s) + w_c * comm(s))$
is minimized

where $proc$ (resp. $comm$) is a function that computes the cost of processing (resp. communicating the result of) a SQL query and w_p, w_c are weights chosen appropriately. In a centralized environment, communication cost might not be relevant in which case, it could be removed from the cost model.

3 Queries and Rewriting Rules

We explore several possible rewritings that share common computation between queries and use the relational optimizer as an oracle to optimize and estimate the cost of individual SQL queries. This technique can be used both by a middle-ware environment (as in [9]) and to extend a relational optimizer to be able to perform the optimizations we are considering.

3.1 Query Definition

In order to better explain the rewriting rules we are using, we first define the query expressions used to build a single XML document. Sorting is omitted in our queries since it does not affect our rewritings.

Definition 1 [Atomic Node] *An atomic node is a node in the DTD which has a unique instance for each distinct instance of its parent node.*

By convention, the root of the XML document is an atomic node. Examples of atomic nodes are `RegName` and `Name` (in the `DTD` of Fig. 1). Each customer has a single value for these nodes. Thus, given a node in the tree, there exists a functional dependency between each instance of that node and each corresponding instance of its atomic children nodes.

Definition 2 [Multiple Node] *A multiple node corresponds to a node in the DTD which may have multiple instances for each distinct instance of its parent node.*

An example of a multiple node is the `PartName` node.

In order to compute instances of an XML node, a unique SQL query is associated with that node. The evaluation of that query results in a set of tuples each of which is used to create an instance. There is a one-to-one correspondence between the tuples that are in the result of a SQL query at a node and the instances of that node. This semantics is similar to that of the Skolem functions used in [9]. Therefore, a key (that might be composed of attributes coming from different relations) is associated with each node in the XML document and is in the set of projected attributes at that node. Each distinct value of the key determines a distinct instance of the node to which that key is associated.

In an XML document, parent/child relationships correspond to key/foreign key joins between parent queries and children queries. Thus, in order to build the XML document tree structure, the query used at a node must always include the query at its parent node. Therefore, queries at nodes are defined as follows.

Definition 3 [Queries] *Given query $Q_p = \pi_{\boldsymbol{p}} exp_p$ at node p ($\boldsymbol{p}$ is the key at node p) and query Q_c at node c, if c is a child of p, then Q_c is defined by one of:*

– $Q_c = \pi_{\boldsymbol{p},\boldsymbol{c}} exp_p$. *$Q_c$ is a simple projection on the expression used to evaluate the parent node p.*
– $Q_c = \pi_{\boldsymbol{p},\boldsymbol{c}}(exp_p \bowtie exp_c)$. *$Q_c$ is a projection on a join expression containing the parent node expression.*

If the relational schema is highly normalized, a join expression is often necessary in computing the instances of XML nodes. In the case of a multiple node, this join is used to build a one-to-many relationship from flat relational data. If the relational schema contains an un-normalized relation, simple projections often suffice to compute node values.

In the definition given above, exp_c can be any expression that might include an arbitrary number of joins. It is necessary that the key value $\boldsymbol{p}$ used to compute the parent node p, is a subset of the keys of its children nodes. An example is the query used to evaluate atomic nodes such as `Name`. In this query, the customer identifier `CUSTKEY` is also projected out and determines the customer to which each name instance is associated. The query used to evaluate the node `RegName` is an example of the presence of a join expression in the child query. In fact, in this particular case, even if a join expression is used, it is guaranteed that there is a unique region name per customer. The expression at the node `PartName` is an example of a query used to compute a multiple node.

Given two queries corresponding to the parent and child elements or to two sibling elements, we rewrite them in two steps. First, these queries are *merged* resulting in a single SQL query. Second, *common sub-expression elimination* is applied to the merged query if it contains redundant expressions. We now define query merging and common sub-expression elimination in our context.

3.2 Query Merging

Definition 4 [Parent/child Merging] *Given a node p and its query Q_p and a node c, which is a child of p and its query Q_c, merging Q_p and Q_c results in a query Q defined as follows:*

1. *If $Q_p = \pi_{\boldsymbol{p}} exp_p$ and $Q_c = \pi_{\boldsymbol{p},\boldsymbol{c}} exp_p$, then $Q = \pi_{\boldsymbol{p},\boldsymbol{c}} exp_p$.*
2. *If $Q_p = \pi_{\boldsymbol{p}} exp_p$ and $Q_c = \pi_{\boldsymbol{p},\boldsymbol{c}}(exp_p \bowtie exp_c)$, then $Q = (\pi_{\boldsymbol{p}} exp_p) ⟕ (\pi_{\boldsymbol{p},\boldsymbol{c}}(exp_p \bowtie exp_c))$.*

An example of the first merge is the case of merging the `Customer` query with the query at its child node `Name`. An example of the second one is the case of merging the `Customer` query with its child node `PartName`.

Definition 5 [Sibling Merging] *Given two sibling nodes c_1 and c_2 sharing a common parent p, merging their queries Q_{c_1} and Q_{c_2} results in a query Q defined as follows:*

1. *If c_1 and c_2 are both atomic nodes such that $Q_{c_1} = \pi_{\boldsymbol{p},\boldsymbol{c_1}} exp$ and $Q_{c_2} = \pi_{\boldsymbol{p},\boldsymbol{c_2}} exp$, then $Q = \pi_{\boldsymbol{p},\boldsymbol{c_1},\boldsymbol{c_2}} exp$.*
2. *If one of c_1 or c_2 is a multiple node, then $Q = Q_{c_1} \cup Q_{c_2}$ where $\cup$ is an outer-union.*

An example of the first kind of sibling merge is the case of nodes `RegName` and `RegCom`. In this case, the expression that merges queries at those nodes has a simple union of the projection lists of the two initial queries. The second sibling merge case is more general. An example of that is the query that results from merging the queries at nodes `PartName` and `SuppName` (see Section 2).

3.3 Exploiting Common Sub-Expressions

Since each query must contain its parent query, the *largest expression* a parent query shares with its children queries is itself. Thus, given two sibling queries, the *smallest expression* these queries have in common is their parent query. Sibling queries could have more in common, though. The query obtained from either the parent/child or sibling query merging will often contain redundant expressions.

The first case of parent/child merging (in Definition 4) and the first case of sibling merging (in Definition 5) rewrite the two input queries in a way that factors out the common expression between the two queries. The two remaining cases, in the same definitions, are the cases that will be discussed in this section.

Definition 6 [Parent/child Sharing] *The merged query is* $Q = (\pi_{\boldsymbol{p}} exp_p) ⟕ (\pi_{\boldsymbol{p},\boldsymbol{c}}(exp_p \bowtie exp_c))$*, where* $(\pi_{\boldsymbol{p}} exp_p)$ *is the parent expression, which can be factored out as follows:* $Q = \pi_{\boldsymbol{p},\boldsymbol{c}}(exp_p ⟕ exp_c)$.

One might think that it is always a good idea to factor out the common parent expression after a parent/child merge. Our experiments, in Section 5, show that this choice might not always be the best depending on the presence of selection predicates in the expressions.

Definition 7 [Sibling Sharing] *Given two sibling queries* Q_{c_1} *and* Q_{c_2}*, where* c_2 *is a multiple node, depending on* c_1 *being atomic or multiple, the merged query* Q *is defined by:*

1. $Q = (\pi_{\boldsymbol{p},\boldsymbol{c_1}} exp_p) \cup (\pi_{\boldsymbol{p},\boldsymbol{c_2}}(exp_p \bowtie exp_{c_2}))$.
2. $Q = (\pi_{\boldsymbol{p},\boldsymbol{c_1}}(exp_p \bowtie exp_{c_1})) \cup (\pi_{\boldsymbol{p},\boldsymbol{c_2}}(exp_p \bowtie exp_{c_2}))$.

The query in the second sibling sharing case is the one that deserves most attention. Let us consider the example of merging `PartName` and `SuppName` (in the `DTD` of Fig. 1) and write the corresponding queries in SQL.

Below, we give three versions of the query that merges $Q_{\texttt{PartName}}$ and $Q_{\texttt{SuppName}}$. `Q`, `MaxQ` and `MinQ` are three equivalent queries where common sub-expressions are treated differently.

```
Q
select  Q.ckey, Q.sname, Q.pname
from
((select distinct 1 as L, C_CUSTKEY as ckey, S_SUPPKEY as skey, S_NAME as sname,
                  NULL as pkey, NULL as pname
  from   CUSTOMER,SUPPLIER,LINEITEM,ORDERS
  where  S_SUPPKEY=L_SUPPKEY and L_ORDERKEY=O_ORDERKEY
  and    C_CUSTKEY = O_CUSTKEY and C_ACCTBAL < 5000 )
 UNION ALL
 (select distinct 2 as L, C_CUSTKEY as ckey,
                  NULL as skey, NULL as sname,
                  P_PARTKEY as pkey, P_NAME as pname
  from   CUSTOMER,PART,LINEITEM,ORDERS
  where  P_PARTKEY=L_PARTKEY and L_ORDERKEY=O_ORDERKEY
  and    C_CUSTKEY = O_CUSTKEY and C_ACCTBAL < 5000)
)  Q
order by Q.ckey,L;
```

`Q` is the “naive” query where $Q_{\texttt{PartName}}$ and $Q_{\texttt{SuppName}}$ are merged using an outer-union. No common expression elimination is applied to it. Dummy field L is introduced to separate parts and suppliers for each customer (for the purpose of building the final XML document).

```
MaxQ
select distinct C_CUSTKEY as ckey, Q.sname,Q.pname
from   ORDERS, CUSTOMER, LINEITEM
```

```
((select 1 as L, S_SUPPKEY as skey,
         S_NAME as sname, NULL as pkey, NULL as pname
  from   SUPPLIER)
  UNION ALL
 (select 2 as L, NULL as skey, NULL as sname,
         P_PARTKEY as pkey, P_NAME as pname
  from   PART)
)  Q
where C_ACCTBAL < 5000 and C_CUSTKEY = O_CUSTKEY
and   L_ORDERKEY = O_ORDERKEY and (Q.skey = L_SUPPKEY or Q.pkey = L_PARTKEY)
order by C_CUSTKEY,L;
```

`MaxQ` is a rewriting of `Q` where the common join between `ORDERS, CUSTOMER` and `LINEITEM` is factored out. This join is the *largest shared expression* between the two siblings. The disjunctive condition `(Q.skey = L_SUPPKEY or Q.pkey = L_PARTKEY)`, results from the fact that the common join expression needs to be joined with suppliers using `Q.skey = L_SUPPKEY` and with parts using `Q.pkey = L_PARTKEY`. The outer-union operation now computes all suppliers and parts.[1]

```
MinQ
select distinct C_CUSTKEY as ckey, Q.sname, Q.pname
from   ORDERS, CUSTOMER,
((select  1 as L, L_ORDERKEY, S_NAME as sname, NULL as pname
  from   SUPPLIER,LINEITEM
  where  S_SUPPKEY=L_SUPPKEY)
  UNION ALL
 (select  2 as L, L_ORDERKEY, NULL as sname, P_NAME as pname
  from    PART, LINEITEM
  where   P_PARTKEY=L_PARTKEY)
)  Q
where C_ACCTBAL< 5000
and   C_CUSTKEY = O_CUSTKEY and Q.L_ORDERKEY = O_ORDERKEY
order by C_CUSTKEY,Q.L;
```

`MinQ` is a variant of `MaxQ` where the main goal is to avoid disjunctive join conditions and cope with current day relational optimizers which are unable to deal with disjunctive predicates efficiently. However, since `MinQ` replicates a portion of the common join condition (in the example, the one with the `LINEITEM` table), it might not always perform better than `MaxQ`. Therefore, we explore both rewritings: *complete common sub-expression elimination* which results in unions and may introduce disjunctive conditions and *partial common sub-expression elimination* which results in unions and has only conjunctive predicates.

4 Optimization

We designed two greedy algorithms: `OptimizeSiblings()` and `OptimizeAll()`. `OptimizeSiblings()` explores merges and computation sharing between sibling queries only while `OptimizeAll()` interleaves merging of sibling queries and merging of parent/child queries.

[1] This may not be the case if additional predicates on parts and/or suppliers are used.

Algorithm 1 `compute_benefit()` Algorithm

```
Require: $x, y$
 1: #define cost(q) (w_p*proc(q)+w_c*comm(q))
 2: c_before_merge = cost(x) + cost(y)
 3: c_after_merge = min(cost(Q(x,y)), cost(MaxQ(x,y)), cost(MinQ(x,y))){pick best sibling
    merge}
 4: benefit(x,y) = c_before_merge − c_after_merge
```

Algorithm 2 `OptimizeSiblings()` Algorithm

```
Require: $Tree$
 1: while not_empty(s_list) do
 2:    pick (x,y): max(benefit(x,y)) in s_list {Pick most beneficial siblings to merge}
 3:    stop if benefit(x,y)<0
 4:    sibling_merge_rewrite(x,y) {replace x, y with merged query}
 5:    children(x)+=children(y) {y-subtree is attached to x}
 6:    remove(y,*) from s_list
 7:    remove(*,y) from s_list
 8:    compute_benefit(x,*) in s_list
 9:    compute_benefit(*,x) in s_list
10: end while
```

4.1 Sibling Optimization

`OptimizeSiblings()`, given in Algorithm 2, explores the benefits of sibling merging for each pair of sibling nodes. The benefit of merging two sibling queries can be either positive or negative. It is computed as the difference between the processing and communication costs of the two queries at nodes `x` and `y` and the rewritten query (where both queries are merged) (see Algorithm 1).

Candidate pairs are stored in a list `s_list`. At each step in the optimization algorithm, the sibling pair that offers the best benefit (say `(x,y)`) is selected to be rewritten. The query at node `x` now contains the merged expression between `x` and `y`. The query at node `y` no longer exists. Thus, all pairs of the form `(y,*)` and `(*,y)` are removed from the candidate sibling merges `s_list`. This includes `(x,y)`, which is no longer a candidate pair. Finally, since the query expression at node `x` has been modified, the algorithm recomputes the benefit of all candidate sibling merges that involve node `x` (i.e., `(x,*)` and `(*,x)`).

Once two sibling queries are merged, `OptimizeSiblings()` rewrites them to eliminate common computation. The algorithm chooses the best of `Q`, `MaxQ` and `MinQ` using `compute_benefits()`.

If two sibling queries are merged, their children queries become siblings and could be considered for additional sibling merges. However, the potential for these new sibling queries to share large common sub-expressions reduces. In addition, considering these queries for sibling merging would increase the search space size. Therefore, as a heuristic, the only candidate sibling pairs `(x,y)` we consider are the ones where `x` and `y` are siblings in the initial set of queries.

Algorithm 3 `OptimizeAll()` Algorithm

```
Require: Tree
 1: while not_empty(list=union(s_list,pc_list)) do
 2:   pick (x,y): max(benefit(x,y)) in list
 3:   stop if benefit(x,y)<0
 4:   if (x,y) in s_list then
 5:     sibling_merge_rewrite(x,y)
 6:   else
 7:     pc_merge_rewrite(x,y)
 8:   end if
 9:   children(x)+=children(y)
10:   remove (y,*) and (*,y) from s_list
11:   compute_benefit(x,*) in s_list
12:   compute_benefit(*,x) in s_list
13:   remove(parent(y),y) from pc_list
14:   compute_benefit(parent(x),x) in pc_list
15:   compute_benefit(x,*) in pc_list
16: end while
```

4.2 Combined Optimization

`OptimizeAll()` is given in Algorithm 3. Once a parent/child or a sibling merge has been performed, the difference between `OptimizeAll()` and `OptimizeSiblings()` is the impact on `pc_list`, the candidate parent/child merges list. Since node `y` does not exist anymore, `(parent(y),y)` needs to be removed from `pc_list`. In addition, since the query at node `x` now contains the merged query, the benefits of `(parent(x),x)` and of `(x,*)` are recomputed. In order to remain within good complexity bounds, the same assumption as for `OptimizeSiblings()` is made on sibling merges. In addition, this assumption is also made for parent/child merges. When a parent/child merge `(x,y)` is performed, the subtree rooted at `y` becomes directly related to `x`. In this case, we do not consider the new children of `x` as candidate merges.

4.3 Cost Analysis

Given $|\mathcal{S}|$ queries, the maximal initial size of `pc_list` is $|\mathcal{S}|$-1=$O(|\mathcal{S}|)$ and the maximal initial size of `s_list` is $\Sigma_{q \in \mathcal{S}}(f(q)(f(q)-1)/2) = O(|S|^2)$, where $f(q)$ is the fanout of query q (number of children queries) in the XML tree. The initialization of the two lists needs $O(|S|^2)$ time and space. At each step where a pair (x, y) is selected, we remove at least one element from `pc_list`. For each node q whose children are in s_list there can be at most $f(q)$ sibling merges each taking at most $O(f(q))$ time. Therefore, the number of iterations is linear in the number of queries and each takes linear time. Thus, the number of steps required is linear in the number of queries: $O(|S|)$ where the initial number of elements in `s_list` is at most $O(|S|^2)$.

5 Experiments

Due to space constraints, we only present a short set of experiments that evaluate sibling rewritings against the rewriting techniques of [9] and [15]. These experiments were carried on a 500Mhz Pentium III PC with 256MB of main memory and refer to an instance of the TPC-R [16] dataset using scaling factor 0.2. We used a commercial RDBMS for storing the data. All tables have indices on primary and foreign keys.

5.1 Data

Our documents conform to a simpler version of the `DTD` in Fig. 1 with only the `Customer`, `PartName` and `SuppName` nodes. There are three basic queries corresponding to the nodes of this DTD. Query `Q1` instantiates the `Customer` node, `Q2` the `PartName` node and `Q3` the `SuppName` node.

We used the field `C_CUSTKEY` of the `CUSTOMER` table to control the size of the documents we build. For the case denoted as "Small-Doc", we instantiated the document for customers with `C_CUSTKEY` less than 5000 (i.e., 5000 tuples). The document denoted as "Large-Doc" is generated with no restrictions on `C_CUSTKEY`. The `Customer` node for both documents is "fat", i.e. all fields from table `CUSTOMER` are published as attributes of `Customer`. Table 1 summarizes the execution times of all possible parent-child/sibling merges as well as the queries corresponding to each node in the DTD. Qij denotes the merged result of queries Qi and Qj. For example `Q12` is the result of a parent/child merge of `Q1` and `Q2`, while `Q23` stands for the sibling merge of `Q2` and `Q3`. The second and third rows of the table are the modified queries (`MaxQ` for complete subexpression elimination and `MinQ` for partial subexpression elimination). Note that common subexpression elimination is not defined for all queries (e.g. `Q1`).

5.2 Results

Looking at the execution times for the Small-Doc case, a first observation is that extensive common subexpression elimination in some cases results in substantially worse performance. The reason for this effect is twofold. The first is that common subexpression elimination might generate disjunctive predicates that are hard to optimize (see query `MaxQ` in Section 3.3). The second reason is that often common expressions are helpful for preserving selections. For instance, both `Q12` and `Q13` make use of the selection on `C_CUSTKEY` through the repeated join with the `CUSTOMER` table. Partial common expression elimination (`MinQ23` and `MinQ123` in this example) is better than complete subexpression elimination but (in the case of `Q123`) no better than no common subexpression elimination. In comparison with the complete common subexpression elimination rewriting, the partial one maintains the join of table `PART` (resp. `SUPPLIER`) with table `LINEITEM` in the rewriting for `Q2` (resp. `Q3`) in `Q23` (same for `Q123`). This is necessary to avoid generating a disjunctive predicate (see `MinQ` in Section 3.3).

In the Large-Doc case, common expression elimination pays off in all cases compared to executing `Q1`, `Q2` and `Q3` independently. This is because the common expression in each merged case is expensive and should be evaluated a minimal number

	Q1	Q2	Q3	Q12	Q13	Q23	Q123
	Small-Doc						
Q	2.91	154.46	200.30	242.86	316.29	568.88	849.07
MaxQ	-	-	-	1034.70	1332.63	473.48	3338.18
MinQ	-	-	-	-	-	349.11	3136.87
	Large-Doc						
Q	11.51	1193.03	1439.83	1396.97	1695.97	3626.64	5897.71
MaxQ	-	-	-	1352.12	1650.23	2366.28	5794.31
MinQ	-	-	-	-	-	2175.90	5586.54

Table 1. Execution Times (secs)

of times. Furthermore, partial elimination of the common subexpression benefits both queries `Q23` and `Q123` as in the previous case.

Looking at the complete times for producing the pieces of the document in the relational engine for all meaningful combinations of the aforementioned queries: [`Q1`, `Q2` and `Q3`], [`Q12` (parent/child merge) and `Q3`], [`Q2` and `Q13` (parent/child merge)], [`Q1` and `Q23` (sibling merge)], [`Q123` (single query)], plan `Q1`+`Q23` is marginally faster than plan `Q1`+`Q2`+`Q3` in the Small-Doc case, while it is about 18% faster in the Large-Doc case.

6 Related Work

Since we are optimizing common sub-expressions among multiple queries, our work share similarities with multi-query optimization (see, e.g., [13]). It is well known that multi query optimization is exponential [13]. Our work benefits from application-dependent information (building XML trees) to optimize sibling queries instead of attempting to optimize an arbitrary subset of queries. This reduces the complexity of the optimization.

In [7], the authors extend relational query engines with a new operator that processes sets of tuples. They define new rewriting rules that involve that operator and show how to integrate that operator in a relational optimizer. This work motivates the necessity to extend relational optimizers. In our work, we do not introduce a new operator, rather, we explore new rewriting rules.

In [9], the authors focus on merging parent and children queries. The rewritings we propose are more general than the ones in [9] and explore an additional dimension that has been proven to result in better efficiency. In [15], the authors provide an extension to SQL to express XML views of relations and carry an experimental study of publishing relational data in XML. This work has not adopted an optimization approach to this problem.

Finally, in [4], the authors present ROLEX, a system that extends the capabilities of relational engines to deliver efficiently navigable XML views of relational data via a virtual DOM interface. DOM operations are translated into an execution plan in order to explore lazy materialization. The query optimizer uses a characterization of the navigation behavior of an application to minimize the expected cost of that navigation. This work could benefit from our new optimizations if they are integrated into a relational system.

7 Conclusion

We discussed the problem of efficiently building XML documents from relations and showed that exploring common computation between sibling queries is a fundamental algebraic rewriting when optimizing SQL queries used to build XML documents. In particular, we showed that in the case where an element has both unique and repeated children, sibling merging combined with partial common sub-expression elimination, enables computation sharing without replicating data. This strategy can be used both inside and outside a relational engine.

References

1. S. Amer-Yahia, M. Fernández. Techniques for Storing XML. Tutorial. ICDE 2002.
2. M. Benedikt, C.Y. Chan, W. Fan, R. Rastogi, S. Zheng, A. Zhou. DTD-Directed Publishing with Attribute Translation Grammars. VLDB 2002.
3. P. Bohannon, J. Freire, P. Roy, J. Siméon. From XML Schema to Relations: A Cost-based Approach to XML Storage. ICDE 2002.
4. P. Bohannon, S. Ganguly, H. F. Korth, P. P. S. Narayan, P. Shenoy. Optimizing View Queries in ROLEX to Support Navigable Result Trees. VLDB 2002.
5. M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. VLDB 2000.
6. D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, M. Stefanescu XQuery 1.0: An XML Query Language. http://www.w3.org/TR/query-datamodel/.
7. S. Chaudhuri, R. Kaushik, J. F. Naughton. On Relational Support for XML Publishing: Beyond Sorting and Tagging. SIGMOD Conference 2003.
8. J. M. Cheng, J. Xu. XML and DB2. ICDE 2000.
9. M. Fernandez, A. Morishima, D. Suciu. Efficient Evaluation of XML Middle-ware Queries. SIGMOD 2001.
10. D. Florescu, D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML in a Relational Database. IEEE, DE Bulletin 1999.
11. Y. Ioannidis. Query Optimization. ACM Computing Surveys, symposium issue on the 50th Anniversary of ACM. Vol. 28, No. 1, March 1996, pp. 121-123.
12. C.C. Kanne, G. Moerkotte. Efficient Storage of XML Data. ICDE 2000.
13. P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. SIGMOD 2000.
14. J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, J. Funderburk. Querying XML Views of Relational Data. VLDB 2001.
15. J. Shanmugasundaram, E.J. Shekita, R. Barr, M.J. Carey, B.G. Lindsay, H. Pirahesh, B. Reinwald. Efficiently publishing relational data as XML documents. VLDB Journal 10(2-3): 133-154 (2001).
16. Transaction Processing Performance Council. TPC-H Benchmark: Decision Support for Ad-Hoc queries. http://www.tpc.org/.