On Relational Support for XML Publishing: Beyond Sorting and Tagging

Surajit Chaudhuri
Microsoft Research
Redmond, WA
surajitc@microsoft.com

Raghav Kaushik*
University of Wisconsin
Madison, WI
raghav@cs.wisc.edu

Jeffrey F. Naughton†
University of Wisconsin
Madison, WI
naughton@cs.wisc.edu

ABSTRACT

In this paper, we study whether the need for efficient XML publishing brings any new requirements for relational query engines, or if sorting query results in the relational engine and tagging them in middleware is sufficient. We observe that the mismatch between the XML data model and the relational model requires relational engines to be enhanced for efficiency. Specifically, they need to support relation-valued variables. We discuss how such support can be provided through the addition of an operator, GApply, with minimal extensions to existing relational engines. We discuss how the operator may be exposed in SQL syntax and provide a comprehensive study of optimization rules that govern this operator. We report the results of a preliminary performance evaluation showing the speedup obtained through our approach and the effectiveness of our optimization rules.

1. INTRODUCTION

While XML is rapidly emerging as a standard for exchanging business data, a large amount of this business data is currently stored in relational databases. Consequently, there has been substantial interest in finding ways to efficiently publish existing relational data as XML [1, 2, 11, 16, 17]. Indeed, all commercial systems have incorporated some support for XML publishing. The approach in all of this work is to define XML *views* of relational data and to issue XML queries over these views. For the most part the focus has been on issues external to the RDBMS, for example, determining the class of XML views that can be defined, studying the languages used to specify the conversion from relational data to XML, and investigating methods of composing XML queries with the XML view.

In this paper, we focus more closely on the class of SQL queries that are typically generated by XML publishing applications, and ask if anything needs to be changed *within* the relational engine to efficiently evaluate these queries. Our answer is yes — due to the difference in the XML and relational data models, some translated XML queries result in relational queries that are awkward to express in SQL and inefficient to evaluate using traditional relational engine query evaluation operators.

The basic issue is that a hierarchical model makes it very convenient and natural to talk about subtrees of a given node in a data set. For example, in a classical part and supplier data set, one might refer to the subtree rooted at a node for a given supplier to conveniently reference that supplier, all the parts it supplies, and all associated information about those parts. Going one step further, in an XML query, it is convenient and natural to perform operations on that subtree, for example, to retrieve all parts in that subtree that cost more than the average price of the parts in that subtree. It is precisely this kind of query, which is natural to express in XML query languages over the hierarchical XML data model, that cause trouble when translated to SQL (when the data originates from an RDBMS).

We do not mean to imply that these queries cannot be translated into SQL. They can; the formal expressive power of SQL is more than sufficient for the task. However, as we will explore in Section 2, with traditional relational technology the resulting queries will be hard to express and inefficient to evaluate. The reason is that the notion of a subtree in XML translates into a set of tuples in the relational model, and classical relational languages and execution engines do not support the notion of binding a variable to sets of tuples and executing subqueries over these sets (we will show that a traditional group by operation is not enough for this purpose).

We propose a three-pronged approach to rendering relational database query processing more effective in XML publishing applications:

1. At the query evaluation level, we identify the need for an operator, which we will call GApply, that binds variables to sets of tuples and allows subqueries to be executed over the relations that are bound to these variables — in other words, an operator that handles relation-valued variables. To the best of our knowledge, the first reference to this operator and its implementation in Microsoft SQL Server 2000 appeared in [12][1]. The notion of binding variables to sets of

*Part of the work done while visiting Microsoft Research

†Supported by NSF grant ITR 0080002

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

[1]It is called SegmentApply in [12]

tuples has also been proposed in [5, 6]. Interestingly, the motivation in this previous work was to support data warehousing applications. In this respect our work adds weight to the claim that such an operator is an important addition to relational query evaluation engines.

2. Even with the GApply operator added to the query evaluation engine, the relational query optimizer will not be able to effectively optimize queries using this operator without useful equivalences or transformation rules that let it consider alternative plans. To attack this problem we propose a number of transformation rules that modify query plan trees containing the GApply operator. This allows the GApply operator to be seamlessly integrated into a Volcano-style optimizer [13]. We note here that since GApply has been identified to be useful in the data warehousing context, all our rules are automatically applicable to decision support queries.
3. Finally, based on our experiments with SQL Server 2000, we argue that it is necessary to expose GApply in the syntax since it is difficult for the parser to detect whether it applies given an arbitrary SQL query. We accomplish this through a minor extension to the SQL language. This explicit representation of the GApply operator in SQL does not change the expressive power of SQL, but it renders XML publishing queries easier to express and, more importantly, makes the need for the GApply operator transparent to the parser. There is some precedent for adding such constructs to SQL; for example, the CUBE and ROLLUP operators do not increase the theoretical expressive power of SQL, but were added for ease of query generation and evaluation.

To get a sense of the efficacy of adding GApply to facilitate the evaluation of XML publishing SQL queries, we performed a preliminary performance study. Our study showed that it is common to get a speedup by a factor of up to 2 times by using our techniques. We also evaluated the effectiveness of our optimization rules in reducing the cost of queries involving GApply, and found them to be effective in optimizing a wide range of queries.

The rest of the paper is organized as follows. In Section 2, we provide examples to motivate our ideas. Section 3 describes the algebraic functionality of GApply. It also describes extensions to SQL to accommodate this operation. The optimization rules are discussed in Section 4. Our performance evaluation is discussed in Section 5. We describe related work in Section 6. Finally, we conclude in Section 7.

2. MOTIVATION

In this section we explore, through examples, the difficulties that arise in expressing and evaluating SQL queries that result from publishing relational data in XML. In such a scenario, the assumption is that the data resides in a relational system; accordingly we have chosen a well-known relational schema for our examples.

Consider the TPCH [9] data. We have reproduced a part of the schema for three tables relevant to us below.

```
supplier(s_key,s_name)
partsupp(ps_suppkey,ps_partkey)
part(p_partkey,p_name,p_retailprice)
```

Next consider an XML view of this data as shown in Figure 1. We use the representation of [1] to associate an SQL query with each node in the schema tree shown. The queries associated with the nodes are shown beside them. Intuitively, we associate a supplier element with each supplier tuple. All parts supplied by a supplier are nested within the supplier element. The parts are bound to the corresponding suppliers through the binding variable $s. Relational attributes can be mapped to sub-elements or attributes. We have omitted some details of the mapping for clarity of exposition. In this paper we assume an unordered model of XML since we believe that this is applicable across a wide spectrum of business applications (in particular, the underlying relational data is not ordered, so it is highly likely that the XML view of this data will not require an ordered model).

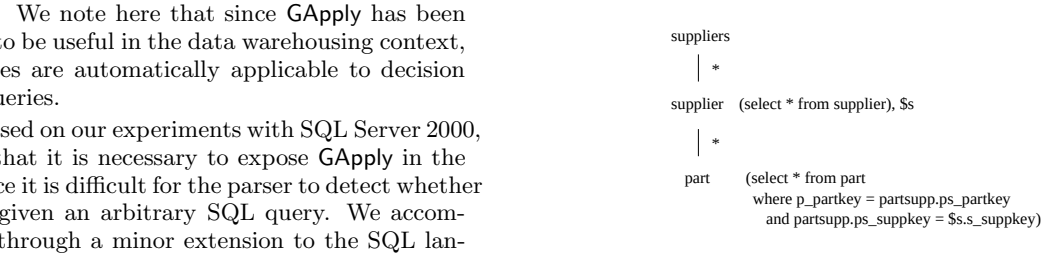


Figure 1: XML view

Now consider the following queries:

Q1: For each supplier element, return the names and retail prices of all parts supplied by that supplier, and also the over-all average retail price of all parts supplied.

Q2: For each supplier element, compute the average retail price of all parts supplied and find the number of parts priced above and below this average.

The above queries perform multiple operations on each supplier element and return their results. In Q1, we return a part of the element along with an aggregate while in Q2, we perform several aggregations over the element.

These queries are readily expressible in XQuery [4]. Q1 in XQuery reads as:

```
For $s in /doc(tpch.xml)/suppliers/supplier
Return <ret> $s/s_suppkey
             <parts>
                 For $p in $s/part
                 Return <part>
                           $p/p_name,
                           $p/p_retailprice
                        </part>
             </parts>
             avg($s/part/p_retailprice)
       </ret>
```

while Q2 reads as:

```
For $s in /doc(tpch.xml)/suppliers/supplier
Return <ret>
        $s/s_suppkey,
        <count_above>
          count($s/part[p_retailprice >=
                   avg($s/part/p_retailprice)])
        </count_above>
        <count_below>
```

```
        count($s/part[p_retailprice <
                avg($s/part/p_retailprice)])
      </count_below>
    </ret>
```

Now assume that these XQuery queries are evaluated by pushing as much of the computation as possible to the server. This is an approach used in projects such as XPeranto to handle XML queries. Thus, a single sorted outer union SQL query [17] is issued to the server to perform the computation and the results are tagged in middleware by a constant space tagger. One way in which query Q1 can be pushed to the server in SQL is:

```
(select ps_suppkey,p_name,p_retailprice, null
 from partsupp, part
 where ps_partkey = p_partkey
 union all
 select ps_suppkey,null,null,avg(p_retailprice)
 from partsupp, part
 where ps_partkey = p_partkey
 group by ps_suppkey)
order by ps_suppkey
```

while one way to push down Q2 would be:

```
(select ps_suppkey, count(*), null
 from partsupp ps1, part
 where p_partkey = ps_partkey and
       p_retailprice >=
             (select avg(p_retailprice)
              from partsupp, part
              where p_partkey=ps_partkey
                and ps_suppkey=ps1.ps_suppkey)
 group by ps_suppkey
 union all
 select ps_suppkey, null, count(*)
 from partsupp ps2, part
 where p_partkey = ps_partkey and
       p_retailprice <
             (select avg(p_retailprice)
              from partsupp, part
              where p_partkey=ps_partkey
                and ps_suppkey=ps2.ps_suppkey)
 group by ps_suppkey)
order by ps_suppkey
```

(We have to order each result by the supplier key since we are assuming a constant space tagger in the middleware [16] and so the result tuples must be clustered by the element to which they correspond. The only way of ensuring this in SQL is by ordering them by the key.)

What we observe in the SQL formulation as opposed to the XQuery formulation is the high degree of redundancy. In XQuery, neither Q1 nor Q2 specifies the "construction" of a supplier element more often than needed. Intuitively, this is because XQuery lets us "refer" to the entire subtree rooted at a supplier by a single syntactic element. In SQL, when we join the partsupp and part tables and group by ps_suppkey, each group defines a (part of a) supplier element. But the group by operation cannot perform anything more sophisticated than an aggregate. Hence, we need to join the partsupp and part tables more often than is conceptually necessary.

A direct evaluation of these resulting SQL queries using a classical set of relational query implementation operators translates to redundant computation within the relational engine. Unfortunately, the fact that such queries can be expressed so naturally in XQuery makes it imperative for relational systems to efficiently support them for XML publishing applications.

For a relational engine to efficiently evaluate this kind of query, we need an evaluation of the above queries that avoids redundant joins between the partsupp and part tables. This means that we must have a handle on each supplier "element" that is constructed dynamically by joining the partsupp and part tables, that is, a handle on the group of tuples that defines a supplier element (this is the relational counterpart of the subtree rooted at the supplier node). Thus, for instance, we should be able to evaluate query Q2 by

1. Joining the part and partsupp tables and grouping by ps_suppkey, where each group defines a (part of) a supplier element, and
2. For each group of tuples, use the handle on the group to iterate over the group to compute the average retail price of all parts supplied and count the number of parts priced above and below this average.

As we will see, the addition of the GApply operator mentioned in the introduction, enables the relational query evaluation engine to find and execute exactly these plans. Our experimental evaluation in Section 5 shows that by this approach we obtain a considerable speedup over current relational implementations. For instance, in our experiments query Q2 runs about twice as fast with GApply than without GApply.

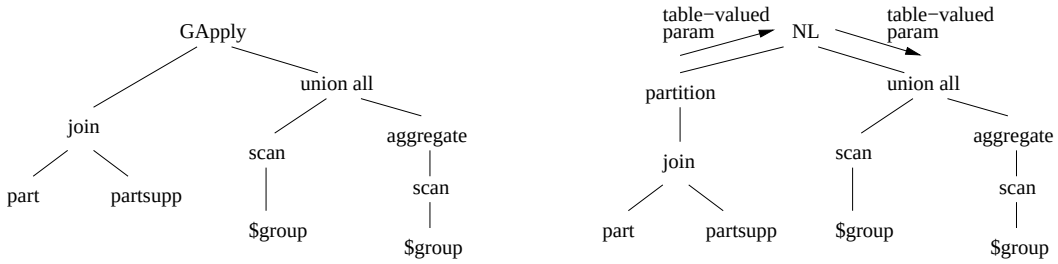
3. THE GAPPLY OPERATOR

What we conclude from the previous sections is that we need a handle on groups of tuples in order to perform operations on those groups. In order to process sets of groups of tuples, we introduce the GApply operator. We now formally describe this operator. The operator GApply($GCols$,PGQ) is specified through a set of grouping (alternatively called partitioning) columns $GCols$ and a query PGQ (short for per-group query). The semantics of GApply are that:

- The input tuple stream is partitioned on the specified columns in $GCols$ to yield a set of groups of tuples.
- The query PGQ is applied on each individual group and a cross product is taken of the result with the value in the grouping columns.
- The output is the union of all the above results taken over all groups.

We refer to the input tuple stream as the *outer* tuple stream and the per-group query as the *inner* query and the roots of the respective queries as the *outer* and *inner* children of GApply. We restrict the power of the per-group query PGQ as follows.

- The only relation it can operate on is the temporary relation associated with the group of tuples under consideration.
- The operators in the per-group query we consider are scan, select, project, distinct, apply, exists, union(all), groupby, aggregate, and orderby. Here, apply is a logical operator that models a subquery [3, 12, 18].



Logical operation Physical operation

Figure 2: Execution of GApply

For instance, query Q1 is represented in our algebra as shown to the left in Figure 2. As opposed to [5], we allow only traditional relational operations in the per-group query. Adding new operators that perform multiple aggregations more efficiently is an orthogonal extension but is not the focus of this paper. Finally, pursuing the terminology of [6], we refer to the operation of GApply as *groupwise processing*.

Formally, the expression $RE_1\ GA_C\ RE_2$ denotes the application of GApply(denoted as GA) on the result of relational algebra expression RE_1 where C is the list of grouping columns and RE_2 is a parameterized relational algebra expression corresponding to the per-group query. The result of this expression is

$$\bigcup_{c \in distinct(\pi_C(RE_1))} (\{c\} \times RE_2(\sigma_{C=c}RE_1))$$

where the union is the union all operation in SQL. Note that we follow multiset semantics. In particular, π does not eliminate duplicates. Instead, we explicitly use distinct for this purpose.

The physical implementation of the GApply operator can be modeled on the way correlated subqueries are currently handled, where a subquery is executed once for each value of the outer tuple. The operator GApply differs in that instead of executing the subquery once for each value of an outer tuple, it executes the inner query for each group of tuples. Thus, we have relation-valued parameters that bind to a set of tuples. The physical implementation takes place in two phases:

- *Partitioning Phase:* The input tuple stream is partitioned based on the values in the grouping columns $GCols$. This can be implemented either through sorting or through hashing.
- *Execution Phase:* This is performed in a nested loops fashion. Each group of tuples is read and the per-group query PGQ is evaluated on each group of tuples. This is achieved by treating each group as a temporary relation, binding a relation-valued parameter $group to each group in succession and passing this parameter to the operator tree for PGQ. When the leaf scan operator receives the relation-valued parameter, it understands this to be a temporary relation and reads from it. After PGQ is executed on a group, its results are returned.

The figure on the right in Figure 2 shows the operator tree for query Q1. Thus, this operation can be implemented with minimal changes to existing relational operators.

We argue that it is difficult for the parser and optimizer to determine when GApply applies. This is particularly so in the presence of the union operation, which is unfortunate since union queries arise very naturally in XML publishing applications. As an example, all of our experiments were run on Microsoft SQL Server 2000 that supports the GApply operator, without however exposing it in the syntax. Although SQL Server succeeds some times in identifying the need for GApply, for the example queries in Section 2, the plans picked by the optimizer do not involve the GApply operator. On the other hand, using the GApply operator in each case considerably speeds up the performance. We thus argue the need to expose GApply in the SQL syntax. This will (a) enable XQuery translators better exploit this functionality in the relational system and (b) help the optimizer identify the GApply operation more easily, especially in the presence of unions.

3.1 Syntax

We propose the following extensions to SQL syntax to handle the incorporation of GApply. The general form of an SQL query performing groupwise processing is:

```
select gapply(PGQ(x)) as <column list>
from <relation list>
where <conditions>
group by <grouping columns> : x
```

The groupby clause has the usual list of grouping columns followed by a special separator ':' which is followed by a single variable name x. Here, x is a relation-valued variable. All columns in the joining tables are associated with x. The select clause has a new key word, gapply. This is followed by a query PGQ on a single table, x.

The associated semantics are that the variable x gets bound to each group in succession, the query PGQ is run on x treating it as a temporary relation, the results are crossed with the value in the grouping columns and the output of the whole query is the union of the results of all these evaluations. The results are clustered by the values in the grouping columns (thus eliminating the need for the order by clause used in Section 2). We allow the result of the query to be bound to a list of column names through the key word as.

The example queries in Section 2 can be written using this syntax as follows. Query Q1 becomes:

```
select gapply(PGQ1(tmpSupp))
```

```
from partsupp, part
where ps_partkey = p_partkey
group by ps_suppkey: tmpSupp
```

where PGQ1(tmpSupp) is:

```
select p_name,p_retailprice,null
from tmpSupp
union all
select null,null,avg(p_retailprice)
from tmp
```

while query Q2 becomes:

```
select gapply(PGQ2(tmpSupp))
from partsupp, part
where ps_partkey = p_partkey
group by ps_suppkey: tmpSupp
```

where PGQ2(tmpSupp) is:

```
select count(*), null
from tmpSupp
where p_retailprice >=
        (select avg(p_retailprice)
         from tmpSupp)
union all
select null,count(*)
from tmpSupp
where p_retailprice <
        (select avg(p_retailprice)
         from tmpSupp)
```

These versions of queries Q1 and Q2 match the corresponding XQuery queries, which suggests that they may be easily generated by automatic query generation tools. With this syntax, we enable the optimizer start off with a plan that has the GApply operator.

We now give our reasons for differing from the syntax proposed in [5].

1. We draw inspiration from the XQuery syntax that allows FLWR expressions in the return clause. The SQL analog of the return clause is the select clause. Hence, we allow subqueries in the select clause.
2. Subqueries in the select clause are already allowed by some systems [12] although with completely different semantics.

Note that while the syntax guarantees that the results are clustered by the grouping columns, the semantics of the GApply operator do not. Hence, the parser should translate a query with the gapply keyword into an operator tree with GApply and a partition operator on top of it. However, if the physical implementation of GApply guarantees the required clustering, then the partition operation is redundant above GApply.

4. TRANSFORMATION RULES

In this section we present transformation rules that enable an optimizer to generate and consider alternatives for query plans containing the GApply operator. All our rules are described in terms of operator trees. In the outer query corresponding to GApply:

1. We assume that the operators are among select, project, scan and join. All leaf nodes are scans while all internal nodes are joins that are annotated with the appropriate selection and projection conditions. Thus, we assume that all selections and projections in the outer query are pushed down. This is basically the annotated join tree representation of [15].

2. We only focus on left-deep join trees where the right child of every internal node is a leaf.

We extend these join trees to allow GApply as an internal node. Before discussing the rules, we present the precise semantics of the operators we consider. We operate under multi-set semantics as standard relational query processors do. Duplicates are eliminated using the distinct operation. The semantics of scan, select, project, join, groupby, union and unionall are as usual (under multi-set semantics), with join referring to inner joins. The apply operator models subqueries and has the following semantics. It takes a relational input R and a parameterized expression E; it evaluates E for each row $r \in R$, and collects the results. Formally,

$$R \mathcal{A} E = \bigcup_{r \in R} (\{r\} \times E(r))$$

where the union is the union all operation in SQL. In the corresponding operator tree, there is a node for the apply operation with an *outer* child that is the root of the operator tree for R and an *inner* child that is the root of the operator tree for E. An aggregate operator takes a relational input R and performs a suitable aggregation (such as sum, count and average) over it. If R is empty, its behavior depends on the exact aggregate being computed. For our purposes, it is sufficient to know that the result is not necessarily empty. Finally, the exists operation takes a relational input R and returns a relation $\{\phi\}$ with one tuple over a null schema if R has atleast one tuple, otherwise it returns the empty relation ϕ. For ease of exposition, we assume that exists can only appear as the inner child of apply. For any relation S, $S \times \phi = \phi$ and $S \times \{\phi\} = S$.

We now present our transformation rules. There are two classes of rules involving GApply.

- Rules that depend on the properties of the per-group query. In order to fire them, we must traverse the operator tree of the per-group query.
- Rules that do not need the per-group query to be traversed. Such rules include:
  - $\sigma(RE_1\ GA_C\ RE_2) = RE_1\ GA_C\ \sigma(RE_2)$ if σ involves only columns returned by RE_2.
  - $\pi_{C \cup B}(RE_1\ GA_C\ RE_2) = RE_1\ GA_C\ \pi_B(RE_2)$

In the rest of this paper, we focus on rules that involve properties of the per-group query. We classify the discussion into three categories as follows. Applying any of the rules we discuss needs cost estimation methods for the GApply operator. We defer a discussion of this to Section 4.4.

4.1 Pushing Computation into the Outer Query

The idea here is to factor out computation from the GApply operation and perform it as part of the outer query.

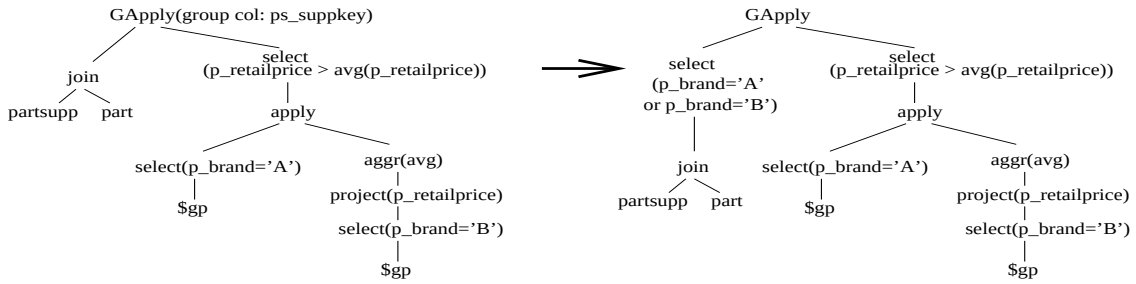


Figure 3: Rule to push selects before GApply

Placing Projections Before GApply

We extract from the outer query, only those columns required by the per-group query, PGQ. Thus, only the grouping columns and those columns referred to somewhere in PGQ need be projected from the result of the outer query. Since the syntax we propose binds all columns of the outer query to the relation-valued variable, this rule can have a significant impact.

Placing Selections Before GApply

If we think of the per-group query PGQ as a function operating on a single input relation x that begins by applying a selection σ on x, then this selection can be applied before the GApply operator. We want to find a minimal subset of the group, in terms of selection conditions, such that running the per-group query on this subset is equivalent to running it on the whole group. For this purpose, we identify the subset of the group that each individual operator needs for the subtree under it to correctly function. We call this selection condition the *covering range* of that operator. The covering ranges are set as follows.

- The covering range of scan is the whole group, defined by the boolean condition true.
- For a select, the covering range depends on its position in the operator tree. If it has an apply, groupby or aggregate descendant, then it is the same as the covering range of its child, otherwise it is the range of its child ANDed together with the condition associated with it.
- The covering range of each of the other unary operators is the same as that of its child.
- The covering range of apply, union and union all is the disjunction of the ranges of the children.

A simple induction on the number of levels in the operator tree of the per-group query leads to the following result.

THEOREM 1. *Let the per-group query be PGQ. Let the temporary relation corresponding to a group be $gp. Let the covering range of the inner child of* GApply *correspond to the condition σ on $gp. Then, $PGQ(\$gp) = PGQ(\sigma(\$gp))$.*

The above theorem suggests that we could push the covering range of the inner child of GApply into the outer query. However, we must be careful with the empty relation. Thus, in the statement of the above theorem, if $\sigma(\$gp) = \phi$, it does not imply that the result of the per-group query is empty — for instance, a count(*) on the empty relation returns a single row, 0. However, if we push σ into the outer query, the per-group query will never be called on the empty relation. Hence, we need to check whether the per-group query PGQ is such that $PGQ(\phi) = \phi$. If this condition is satisfied, then the covering range of the inner child of GApply can be pushed into its outer query. Any selection in the operator tree of the per-group query that is logically equivalent to the covering range of the root can then be eliminated. We check that $PGQ(\phi) = \phi$ by traversing the operator tree bottom up and setting a bit emptyOnEmpty for each node in the tree indicating whether the tree rooted at that node produces an empty output on an empty input.

- For scan, emptyOnEmpty is true.
- For select, project, distinct, groupby, orderby and exists, emptyOnEmpty is the value of emptyOnEmpty of the child node.
- For aggregate, emptyOnEmpty is false.
- For apply, emptyOnEmpty is the value of emptyOnEmpty of the outer child.
- For union and unionall, emptyOnEmpty is true if and only if emptyOnEmpty is true for all the children nodes.

The rule states that:

$$RE_1 \; GA_C \; RE_2 \quad = \sigma_{\text{covering-range}(RE_2)}(RE_1) \; GA_C \; RE_2$$
$$\text{if } RE_2(\phi) = \phi$$

The selection that is inserted on top of the outer tree can then be pushed down using the traditional rules for doing so.

Example: One simple instance of this rule is shown in Figure 3. The operator tree to the left is the one before the rule is fired. It tries to find for each supplier, all parts of brand A that are priced above the average price of parts of brand B. Through the above rule, we get the operator tree on the right. This extracts information only pertaining to parts of brand A and B for each supplier.

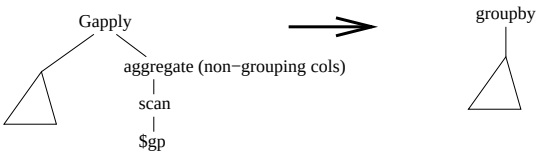


Figure 4: Rule to convert GApply into groupby

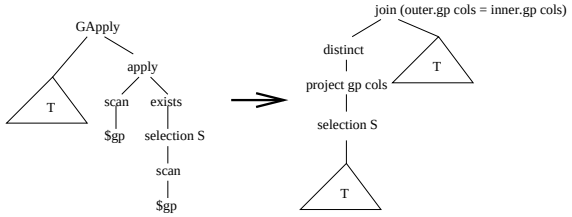


Figure 5: Rule to handle group selection

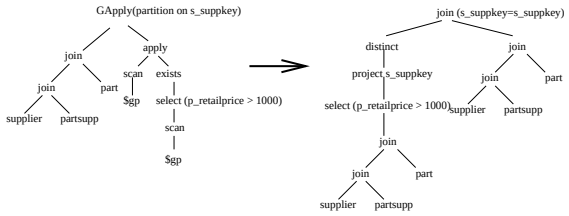


Figure 6: Example of this rule

Converting GApply *to* groupby

If all the per-group query does is to compute aggregates on non-grouping columns, the GApply operator can be replaced with a groupby. The set of grouping columns is the same as the set of partitioning columns for GApply. The aggregates returned from the groupby are the same as the aggregates returned from the per-group query. With a little care, this can be extended even if the aggregate is on grouping columns. This rule is illustrated in Figure 4.

Similarly, if all the per-group query does is scan the group and group by a set of columns B, then GApply can be replaced with a groupby where the grouping columns are $C \cup B$ (here, C is the set of partitioning columns of GApply) and the aggregates are the ones in the per-group query. The above rules when applied in conjunction with the rule involving selections can lead to many transformations.

4.2 Group Selection

We now consider a class of queries that, if we think of each group as representing a complex object, select objects based on expensive predicates. The per-group query either returns the whole group or nothing at all based on some predicate. This is conceptually the case with XPath [8] queries that return the whole sub-tree under a node based on say, an exploratory search within the tree.

Let us consider the example query on the XML schema in Figure 1 that finds all suppliers that supply *some* expensive part. Written in XQuery, this query reads as follows.

```
For $s in /doc(tpch.xml)/suppliers
          /supplier[/part/p_retailprice > 1000]
Return $s
```

There are two ways of evaluating this query. One is to join suppliers with parts, group by the supplier key, suppkey (i.e., construct all supplier objects), and for each supplier key, check whether the group-selection predicate is true by evaluating a query on its group and if so, to return the group. This corresponds to an evaluation of the above query involving GApply where suppkey is the grouping column. The other alternative involves two phases, the first being one where the selection condition is used merely to extract the supplier *keys* followed by one where the groups associated with these keys are constructed by evaluating the necessary joins again. Clearly, if the predicate is highly selective, the latter strategy will win.

The above is formulated as a rule shown pictorially in Figure 5. The operator tree to the left has a GApply instance. Its outer child is the root of the operator tree T. The per-group query checks for the existence of some tuple satisfying selection condition S. If the condition is satisfied by some tuple in the group, the whole group is returned. Otherwise, nothing is returned. This operator tree is transformed to the one shown on the right. This operator tree represents the following evaluation. The outer query T corresponding to GApply is evaluated first with the selection condition S attached to it. If we now project the partitioning columns of GApply, what we have is the set of "group ids" since the id of a group is defined by the values in the grouping columns. However, we do not necessarily have the rest of the group. As a result, we have to join these group ids with the operator tree T to reconstruct the groups. The join condition is the natural equijoin on the grouping columns. Since we need to ensure that each group id is produced only once (for multiset semantics), we add a distinct operator. An application of this rule to the above example is shown in Figure 6.

We next consider object selection based on aggregate conditions. For example, consider the query (on the schema in Figure 1):

```
For $s in /doc(tpch.xml)/suppliers/supplier
Where avg($s/part/p_retailprice) > 10000
Return $s
```

As in the other query above, this query can be evaluated in two ways. We can construct the supplier groups first and then check the aggregate condition, or push down the condition to extract qualifying supplier keys and then construct the respective groups. The question arises what is the advantage at all of the latter plan. The key is to note that if GApply were to be implemented by hashing on the supplier key s_suppkey, a lot of memory would have to be managed whereas if for each supplier, all we store is a sum and a count (which are stored while computing the average) the space required is all of a sudden less. Moreover, computing an average per supplier can be performed in a pipelined manner. Coupled with this, if the condition is selective, then the latter plan is likely to be cost-effective. This rule can be formulated in a manner analogous to the previous rule.

4.3 Pushing GApply below joins

Finally, we consider a generalization of the rules for optimizing groupby in relational systems [7, 14, 19, 20]. The idea there briefly is to think of groupby as an operator and push it below joins, the motivation being that groupby reduces the number of output tuples considerably since it outputs one tuple per group. Rules to pull groupby above joins are also discussed in this body of work. The same motivation also applies to GApply since as a special case, GApply can return one tuple per group (for instance the one having the minimum value on some column). A rule to pull GApply above a join is proposed in [12]. We consider the problem of pushing GApply below a join which unlike the above, involves an analysis of the operator tree of the per-group query.

For definiteness, we pick the invariant grouping transformation rule [7] for pushing groupby below joins and show

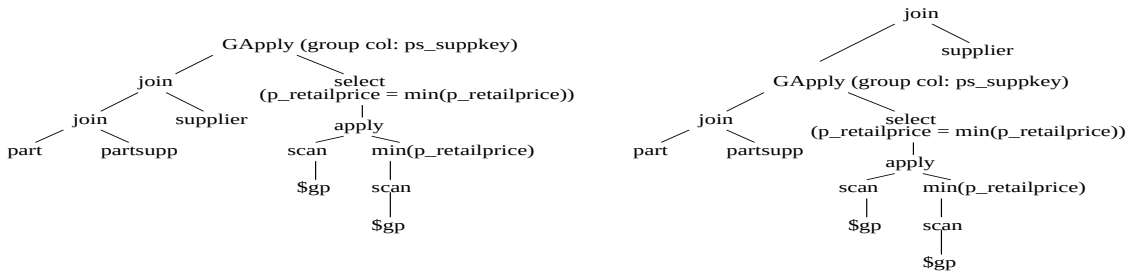


Figure 7: Example of invariant grouping transformation

how this can be extended to pushing GApply below joins. Other rules to push groupby below joins such as simple coalescing grouping can be extended analogously.

Definition 1. [7] The *join columns* of a node n are all columns that participate in join predicates evaluated at ancestors of n. *Required columns* of n are its join columns together with the grouping columns of the query.

Let us now define the notion of *group-evaluation (gp-eval) columns*. Intuitively, gp-eval columns are those columns that are needed to evaluate the per-group query. This includes all columns over which there is a selection predicate. However, with projected columns, unless they are aggregated, they are not gp-eval columns since they could potentially be obtained by performing joins later. More formally, for each operator o in the per-group query we associate a set of columns — call them eval columns — that correspond to the columns needed by the operator tree rooted at o.

- For a scan, this is the empty set
- For a select, it is the union of the eval columns of the child of o with the columns over which there is a selection condition at o.
- For a groupby, it is the union of the eval columns of the child with the grouping columns of o and the columns returned from o.
- For aggregate and orderby, it is the union of the eval columns of the child with the corresponding aggregated/ordering columns.
- For other unary operators, it is the eval columns of the child.
- For apply, it is the union of the eval columns of both children.
- For union and unionall, it is the union of the eval columns of all children.

Finally, the gp-eval columns of the per-group query are the eval-columns of its root.

If the GApply operator is to be placed above a node n in the left-deep join tree, then it must be the case that n has the gp-eval columns. However, the per-group must be modified to *project* only columns available at n. Other columns presumably get attached later through joins. This can be achieved by simply eliminating the columns not available at n from all project lists. We call the modified per-group query the *adapted* per-group query. It is conceivable that at some node o in the per-group query, the only columns projected are those not available at n. This can only happen in a subquery where the apply condition is an exists or not exists where the projected column does not really matter. We avoid this case if we assume without loss of generality that such subqueries project only grouping columns.

Let us now consider when the whole GApply operation can be executed above a given node n, instead of at the top of the (left-deep) join tree. The GApply node must first of all make sense above n, which means that all grouping columns and gp-eval columns must be present at n. The joins above n should also make sense when the GApply instance is placed above n. In addition, for the evaluation of GApply to be correctly preserved under multi-set semantics, all joins above n must be foreign-key joins — a join is a foreign-key join if the join condition is a key foreign-key equality condition where the outer (left) child has a foreign key to the inner (right child) in the left-deep tree.

Definition 2. A node n in the given left-deep tree has the *invariant grouping* property if

1. The columns present at n contain the grouping columns and the gp-eval columns.
2. Every join column of n is a grouping column of the query.
3. Every join above n is a foreign-key join.

If node n satisfies the invariant grouping property, then we can then move GApply above n.

THEOREM 2. *If a node n in a left-deep tree T has the invariant grouping property, then the extended left-deep tree T' obtained by moving the* GApply *node as the parent of n along with the adapted per-group query is equivalent to T.*

Note that if n has the invariant grouping property, so do its ancestors. Thus, GApply can be moved on top of any of n's ancestors.

Consider for instance a query that finds for each supplier, the supplier name and the part that is least expensive. This query can be evaluated in groupwise fashion for each supplier key. The plan is shown on the left in Figure 7. Now the top most join is a foreign key join with the supplier table. Also, to define a group and compute the least expensive part, we do not need any supplier details except the supplier key. Thus, the whole groupwise processing can be pushed below this join to yield the plan to the right in Figure 7.

We note again that other rules to push groupby below joins can be similarly generalized to GApply.

4.4 Integrating the Rules into an Optimizer

In order to integrate the above rules into a Volcano-style rule-based optimizer, we must address the following issues.

First of all, the above rules either push GApply down in the join tree, or altogether eliminate GApply, or add new selections and projections in the outer subtree corresponding to GApply, none of which can be reversed by any of the other rules. Hence, successive firing of rules will terminate.

Secondly, the optimizer must be able to estimate the cost of the GApply operation. We sketch how this can be achieved.

With a uniformity assumption on the groups, we can estimate the cost of GApply as the cost of evaluating the per-group query on one group multiplied by the number of groups. The number of groups is the number of distinct values in the grouping columns. In order to estimate the cost of evaluating the per-group query on a single group, we need to know the size of the group and also need statistics on the group for purposes such as selectivity estimations. We can obtain the size by observing that the average size of a group is the result size of the outer query divided by the number of groups. As for the statistics, since we assume uniformity among the groups, implying for instance that the selectivity of a predicate is the same in all groups, all we need to do is obtain statistics on a single group. A single group can be defined by placing a selection condition on the grouping columns fixing them to a certain value. Now, we have reduced the problem to the traditional problem of obtaining statistics for the result of a (traditional) query.

5. PRELIMINARY EXPERIMENTS

The goals of our performance study are two-fold. Firstly, we wish to get a sense of the efficacy of the GApply operator in speeding up queries. Secondly, we wish to understand the impact of each of our transformation rules. Our experiments are run on Microsoft SQL Server 2000 that supports GApply, without exposing it in the syntax. Since for our experiments we need control over the invocation of GApply, we simulate the operation of GApply on the client side.

5.1 Client Side Simulation of GApply

As explained in Section 3, GApply operates in two phases: Partition and Execute. We describe how each of these phases is simulated on the client side. The partition phase can be implemented either by sorting or hashing the input on the grouping columns of GApply.

- The sorting alternative can be implemented by adding an order by clause to the outer query.
- For the hashing alternative, what we want to simulate is the cost of reading the input tuple stream from the outer query, hashing each tuple on the grouping columns, managing main memory resources in this process. Note that this cannot be performed directly on the server since such an operation is not exposed in SQL syntax — in particular, a simple groupby only maintains aggregates discarding the actual values. However, a groupby that counts the number of *distinct* values in some column has to maintain the actual values to check for distinctness. We exploit this fact to perform our simulation. We create a temporary table, tmpTable, that stores the result of the outer query and wish to scan this table, group by the grouping columns and count the number of distinct values in all non-grouping columns. However, SQL does not allow a count distinct operation on more than one column. To circumvent this problem, we arrange for tmpTable to have the grouping columns with a *single* extra column, miscCols, that stores the result of concatenating the values in all non-grouping columns. We then run the query $Q_{partition}$:

```
select <grouping cols>,count(distinct(miscCols))
from tmpTable
group by <grouping cols>
```

This query needs to partition the tmpTable on the grouping columns while at the same time retaining the column value of miscCols, since it needs to check distinctness. By forcing all miscCols to be distinct (by performing a bit-xor with a counter that is incremented for each row), we force all miscCols values to be managed by the server. Our argument is that this operation simulates the partition phase of GApply. The above computation however performs the extra task of hashing the miscCols value and checking distinctness through string matching. These are parts that are not present in the real partition phase we wish to simulate. We estimate the extra work done as the CPU effort spent on the query $Q_{overestimate}$:

```
select count(distinct(miscCols))
from tmpTable
```

Next, we come to the execution phase. This is simulated in the following manner. We store the result of the outer query in another table without disturbing the columns this time. For each distinct value in the grouping columns, we extract an appropriate range of this temporary table defined by the grouping columns into another temporary table and run the per-group query on the latter table. This is representative of the work done in the execution phase.

5.1.1 Estimating Running Time

We measure both the elapsed time and the CPU time since our queries are usually CPU bound. We consider operator trees where GApply is the top most operator. Such queries involve three parts: (1) running the outer query, (2) partition phase of GApply and (3) execution phase of GApply. The time involved in running the outer query is obtained directly by running it on the server.

If we partition by sorting on the grouping columns, the time taken for this can be measured by forcing the outer query to order the results by the grouping columns. For the hashing alternative, accounting for the partition and execution times is more subtle. The elapsed time for $Q_{partition}$ includes not only the cost of hashing the output of the outer query, but also cost of retrieving whatever buckets spill to disk. Having fetched the buckets however, $Q_{partition}$ moves on to count the number of distinct values in miscCols. What GApply is supposed to do is to retrieve the buckets into memory and perform the *execution phase*. Thus, we only count the CPU time of running the per-group query on each group. This is added to the time taken by $Q_{partition}$, and the CPU time taken by $Q_{overestimate}$ is subtracted to yield the total time for the partition and execute phases.

We argue that the above over-estimates the cost of the execution of GApply. The reason is that while every component of a server side processing is accounted for, our simulation is blocked. For instance, with a hash-based partitioning, we include the time for returning *all* results of the outer query and also the time for reading *all* these results back from tmpTable. In addition, running a query over every group and measuring the total time by adding individual running times will add up the per-query overheads those many times. For all these reasons, we expect that the real elapsed time taken by the full server side implementation will be less than

the time taken by our simulation. As we will see in the next section, for the one query where the server picks a plan involving GApply, the corresponding client-side simulation takes 20% more time.

5.2 Results

Experimental Setup

The data set we use is the TPCH benchmark data. We use the 5GB database. The server has a processor speed of 1GHz while the main memory size is 784MB. The buffer pool size is set to 512MB. Each query is run several times and the average time is taken. All results are based on a cold buffer pool. As discussed above, we measure both the elapsed and CPU times.

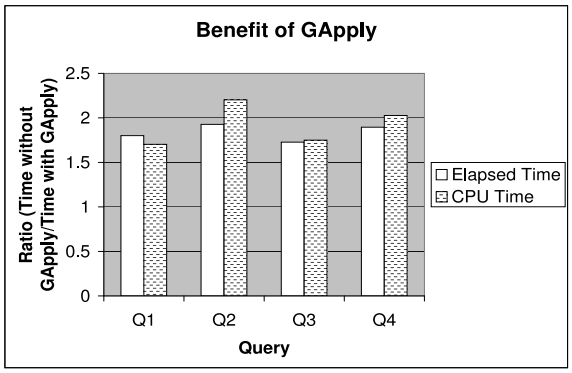


Figure 8: Speedup using GApply

Effectiveness of GApply

We pick a set of queries including the ones presented in Section 2. We run them with and without GApply. Figure 8 shows the results. The X-axis shows the queries. The Y-axis shows the ratio of the time taken without GApply to the time taken with GApply. Thus, a ratio of 2 indicates 50% speedup. Queries Q1 and Q2 are the ones from Section 2.

- Query Q3 also involves a union of two queries intended to represent multiple operations over a supplier element. Stated in English, Q3 finds for each supplier, all part names and prices where the prices are high-end or low-end, where a price is high-end if it is more than a certain fraction of the maximum and low-end if it is less than a certain multiple of the minimum.
- Query Q4 is: for each supplier, for each part size supplied, compute the average retail price and find all parts with this size priced more than the average.

In our experiments, the impact of GApply is comparable whether we perform partitioning through sorting or through hashing. What is shown in Figure 8 is the speedup obtained through hash-based partitioning. Let us recall that the system we are using has the GApply operator implemented. Query Q4 is one where we are able to force the use of this operation in the server. It yields us a wholly server-side experiment to evaluate the impact of GApply. Q4 is expressed in SQL as:

```
select tmp.ps_suppkey,p_name,p_size,p_retailprice
from (select ps_suppkey, p_size, avg(p_retailprice)
      from partsupp, part
      where p_partkey = ps_partkey
      group by ps_suppkey, p_size)
        as tmp(ps_suppkey,p_size,avgprice),
      partsupp, part
where ps_partkey = p_partkey and
      partsupp.ps_suppkey = tmp.ps_suppkey and
      partsupp.p_size = tmp.p_size and
      p_retailprice > tmp.avgprice
order by tmp.ps_suppkey
```

In this case, the optimizer correctly identifies this to be an opportunity for using GApply and this query is executed in the following manner — join partsupp and part, group by ps_suppkey and p_size, and iterate over each group to compute the average and the parts priced more than the average. Query Q4 thus provides us with an opportunity to test the overhead of client-side simulation. The client side simulation of Q4 takes 20% longer than the server-side implementation. Not surprisingly, writing query Q4 in a different but semantically equivalent manner yields a plan that takes orders of magnitude longer to execute than the plan using GApply, just as writing a CUBE query using group by is likely to perform poorly. This is further evidence of the need for syntactic support for GApply.

The main conclusions to be drawn from the above graph are:

- GApply is a useful operator even in the context of simple XQuery queries, such as the ones in Section 2, yielding improvements of factors of up to 2 times faster.
- The queries picked are representative of a wide class of queries. Hence, the operator has a potentially wide applicability.
- The performance improvements for queries Q1, Q2 and Q3 are likely to be even better if we deploy the server side implementation of GApply. This is so since our simulation from the client side is conservative, as evidenced by the results for query Q4.

Effectiveness of Optimization Rules

We determine the effectiveness of each optimization rule by testing the improvement obtained by firing the rule. Our methodology is the following. For each rule, we pick a relevant parameterized query. We then vary the parameter and for each of its values, find the performance benefit obtained by applying the rule. The performance metric is elapsed time.

For instance, consider the transformation rule that handles group selection predicates, described in Figure 5. We test the effectiveness of this rule in the following manner. We pick the parameterized query:

```
For $s in /doc(tpch.xml)/suppliers
          /supplier[/part/p_retailprice > x]
Return $s
```

where the parameter x determines the selectivity of the selection condition. We then vary x to obtain a series of selectivity values for each of which we find the benefit of applying this rule. Table 1 shows the results.

The benefit is measured as the ratio of the (elapsed) time taken without the rule application to the time taken after

Rule Class	Rule	Maximum Benefit	Average Benefit	Average over Wins
Basic Rules	Placing Selection Before GApply	732.94	124.97	124.97
	Placing Projection Before GApply	5.05	3.42	3.42
	Converting GApply To groupby	1.3	1.19	1.19
Group Selection	Exists	14.6	1.67	1.93
	Aggregate Selection	6.3	2.08	3.72
GApply and Joins	Invariant Grouping	2.56	1.32	1.32

Table 1: Effect of Transformation Rules

the rule is fired. The maximum benefit indicates the best performance improvement among the experiments we conducted. The "average over wins" column indicates the average benefit of firing a rule when it actually lowers the cost.

The main observations to be made are:

- Over all, the rules we propose can have a significant impact on the elapsed time of a query involving GApply.
- There are some rules that always lower the cost of the query, indicated by equal values in the "average benefit" and "average over wins" columns. For the other rules, depending on the query, firing the rule can have a positive or negative impact on cost.
- The benefit of converting GApply into groupby is comparatively lower. This is not surprising since the amount of work performed by a GApply when it aggregates non-grouping columns is the same as the work performed by groupby. However, GApply is blocked. Hence, the conversion to groupby helps.

6. RELATED WORK

There has been lots of work in the area of publishing relational documents as XML. As part of the Xperanto [2] project, in [17], an extension of SQL is proposed to specify the construction of XML documents from relational data and several ways of implementing this conversion are discussed. The conclusion there is that in most cases, it is best to push as much computation as possible into the relational engine. Further, in [16], the problem of executing XML queries over the XML view of relational data is considered. They describe how to compose an XML view definition with the XML query and push down computation into the relational engine. The SilkRoute [11] project also a language to specify the conversion between relational data and XML. In [10], several alternatives are considered between pushing all the computation as a single query into the relational engine and issuing several smaller queries. They provide a greedy heuristic algorithm to be implemented in middleware that explores several alternatives between these extremes in a cost-based manner and returns the cheapest alternative. In addition, a view composition algorithm to compose XML queries over an XML view is proposed. All these implementations do not modify the set of relational operators. In the ROLEX project [1], the authors observe that at the end of executing an XML query over a relational database, the resulting XML fragment is then parsed by the application. To overcome this inefficiency, they change the world view so that relational engines return a navigable result tree. When the query results are navigable, the navigation patterns vary across applications. To accommodate this, they modify a traditional cost-based optimizer to be cognizant of the navigation profiles of applications. However, again, the traditional set of relational operators is unchanged.

Our work differs from all of the above in asking whether the whole process of XML publishing has any impact on the core relational operators and answering the question in the affirmative. In setting out our syntax, we draw inspiration from the syntax proposed in [17] where subqueries in the select clause of SQL are used.

In the context of data warehousing, earlier work [5, 6] has identified the need for what is referred to as *groupwise processing* of tuples for data warehousing applications. Briefly the idea is to address queries that intuitively partition the data into groups of tuples and perform repeated computations on them. Expressing this simple intuition in SQL is cumbersome and the authors of [5] propose extensions to SQL to handle this along with a new operator to perform groupwise processing. In [6], the authors adopted the alternative approach of syntactically processing a SQL query to identify whether groupwise processing was possible. The above papers do not discuss how this operation affects the optimization framework and how this operator interacts with traditional relational operators. In [12], the implementation of an operator that performs groupwise processing is described. As noted in Section 3, this is the GApply operator we recommend. We also propose a syntax to expose this operator in the syntax and Section 3 discusses why and how how our syntax is different from the one proposed in [5]. Two rules for optimizing an operator tree in the presence of GApply are presented in [12] one of which is identified in Section 4. The other rule describes when the optimizer considers the use of GApply, given an operator tree without

GApply. We study the impact of GApply on query optimization in detail to complete the picture.

Some of our transformation rules are a generalization of the rules involving groupby in the traditional setting [7, 14, 19, 20]. Thus, our rules to push GApply below joins generalize similar operations on groupby, addressed in [7, 19] while the rules to pull GApply above joins generalize the "lazy aggregation" technique proposed in [20].

7. CONCLUSIONS

In this paper, we asked what impact the requirement for efficient XML publishing had on the core relational engine. Our answer was that relational engines must provide support for binding variables to sets of tuples as opposed to single tuples. We described how such support may be enabled through the GApply operator with seamless integration into existing relational engines. We discussed how this operator can be exposed in the syntax and studied a series of optimization rules that govern the interaction between this operator and traditional relational operators. Our performance study underlined the effectiveness of our techniques. We believe our study provides insight into a fundamental difference between the XML and relational data models.

There are several problems left open for discussion.

1. How should the modified syntax be exploited by algorithms to translate XML queries over XML views of relational data?

2. Are any other changes necessary within the relational query engine to meet the requirement of XML publishing?

3. What about changes to the query optimizer? If we assume that the relational database returns navigable results, then the work in [1] addresses this question. However, if we assume we are running "sorted outer union" SQL queries, this question is open.

We hope to address them in future work.

8. REFERENCES

[1] P. Bohannon, S. Ganguly, H. Korth, P. Narayan, and P. Shenoy. Optimizing view queries in ROLEX to support navigable result trees. In *Proceedings of VLDB*, 2002.

[2] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB*, 2000.

[3] P. Celis and H. Zeller. Subquery elimination: A complete unnesting algorithm for an extended relational algebra. In *ICDE*, 1997.

[4] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery: A query language for XML. World Wide Web Consortium, `http://www.w3.org/TR/xquery`, Feb 2000.

[5] D. Chatziantoniou and K. A. Ross. Querying multiple features of groups in relational databases. In *VLDB*, 1996.

[6] D. Chatziantoniou and K. A. Ross. Groupwise processing of relational queries. In *VLDB*, 1997.

[7] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, 1994.

[8] J. Clark and S. DeRose. XML path language (XPath) 1.0. W3C recommendation. World Wide Web Consortium, `http://www.w3.org/TR/xpath`, Nov. 1999.

[9] T. P. P. Council. TPC-H: Decision support for adhoc queries. `http://www.tpc.org`.

[10] M. Fernández, A. Morishima, D. Suciu, and W. Tan. Publishing relational data in XML: The SilkRoute approach. In *IEEE Data Engineering Bulletin*, 2001.

[11] M. Fernández, D. Suciu, and W. Tan. SilkRoute: Trading between relations and XML. In *WWW9*, 2000.

[12] C. A. Galindo-Legaria and M. M. Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, 2001.

[13] G. Graefe and W. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[14] A. Gupta, V. Harinarayan, and D. Quass. Aggregate query processing in data warehousing environments. In *VLDB*, 1995.

[15] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[16] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, 2001.

[17] J. Shanmugasundaram, E. Shekita, R.Barr, and M. Carey. Efficiently publishing relational data as XML documents. In *VLDB*, 2001.

[18] Q. Wang, D. Maier, and L. Shapiro. Algebraic unnesting of nested object queries. In *Technical Report, Oregon Graduate Institute*, 1999.

[19] W. P. Yan and P. A. Larson. Performing group-by before join. In *ICDE*, 1994.

[20] W. P. Yan and P. A. Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.