Optimizing DOM Programs on XML Views over Existing Relational Databases

Atsuyuki Morishima[1] and Akira Kojima[2]

[1] Research Center for Knowledge Communities, University of Tsukuba
amorishima@acm.org
[2] Graduate School of Engineering, Shibaura Institute of Technology
m103197@sic.shibaura-it.ac.jp

Abstract. Since XML has become the de-facto standard for data interchange through the Internet, more and more application programs to process XML data are being developed. On the other hand, a huge amount of existing data is stored in relational databases. We developed a system that allows XML application programs using DOM APIs to take as input relational data through their XML views. A key issue is optimization. We developed an optimization technique having a training mechanism for efficient execution of the programs. This paper presents the system's architecture, the optimization technique, and preliminary experimental results showing that the proposed technique can achieve dramatic performance improvements.

1 Introduction

Since XML has become the de-facto standard for data interchange through the Internet, more and more application programs to process XML data are being developed. On the other hand, a huge amount of existing data is stored in relational databases. One possible reason is that XML is a *data format* while the relational database provides a *data model* and facilities for efficient data management. In order to connect the two different worlds, how to process *XML views* of databases has been a crucial research topic.

An XML view is *virtual*, if the system materializes a part of the XML view only when an application consumes the part. In other words, it materializes only a required part of the data on demand. The virtual XML view approach has the following two advantages compared to full materialization of XML views: First, it is *scalable*. It does not require a huge amount of additional memory or storage to keep the entire XML data at a time. In general, the data stored in the underlying databases can be huge in size, and full materialization of XML views are impractical in such cases. Moreover, application programs may not need the entire XML data but only a small portion of it. Second, we do not have to worry about the freshness of data and can assume that the obtained data are always up-to-date. The approach is especially effective in situations where the target databases are autonomously managed and actively updated.

A. Persson and J. Stirna (Eds.): CAiSE 2004, LNCS 3084, pp. 248–262, 2004.
© Springer-Verlag Berlin Heidelberg 2004

Technically, the (virtual) XML view approach has to translate operations of XML data into that of underlying databases. Previous works [2][5] developed techniques to translate declarative XML queries, such as XQuery queries, into SQL queries.

We focus on the problems of how to allow XML application programs using DOM [8] API (shortly, *DOM Programs*) to take as input relational data through their XML views and how to optimize them. Since both XQuery-like declarative queries and DOM-style operations are major ways for XML processing, frameworks to allow DOM programs to process XML views have huge benefits. In particular, if we got such a framework, any existing application program to process XML data through DOM API could process any data stored in relational databases without the source code changed. We do not have to redevelop nor rewrite such application programs in order to process the data in databases. This is a significant advantage, especially when one of the important concerns is the software development cost.

An important technical problem of XML view approaches is *optimization*, since naive translation of XML operations into SQL queries can lead to inefficient executions. Efficient evaluation of XML views of relational data have been discussed in situations where XML operations are written in XQuery-style declarative query languages [2][5].

However, it is a non-trivial task to develop optimization techniques to work with DOM programs; A typical DOM program contains a large number of DOM operations scattered over the code. A naive execution framework, in which one SQL query is generated and executed for each DOM operation, is obviously inefficient, since it requires a large number of SQL query submissions. One might consider methods to compose more than one DOM operations into one SQL query. The DOM operations, however, are interleaved with other code written in ordinary, procedural programming languages, such as java, and the control of executions does not depend on the DOM operations only. This makes it difficult to apply normal optimization techniques for declarative queries to the problem.

This paper proposes an optimization framework for DOM operations, interleaved with procedural programming language codes, to deal with XML views of relational data. The underlying idea is to observe executions of DOM programs in order to find efficient *SQL generation rules* for later executions of the programs. The SQL generation rules are used to construct a fewer number of SQL queries having the same effects with less execution costs compared to the naive SQL generation method.

Figure 1 shows the architecture of our system. First, the user has to give to the system a *view query* to define an XML view of relational data stored in the relational database. Then, ordinary DOM programs whose inputs are XML data can be applied to the relational data. Note that the XML view is *not* materialized. Instead, results of DOM operations are computed by SQL queries on demand. The rule repository stores SQL generation rules, which describe *when and what SQL queries to generate and execute*. In the first execution, every DOM operation is translated into one SQL query, as explained in Section 2. During

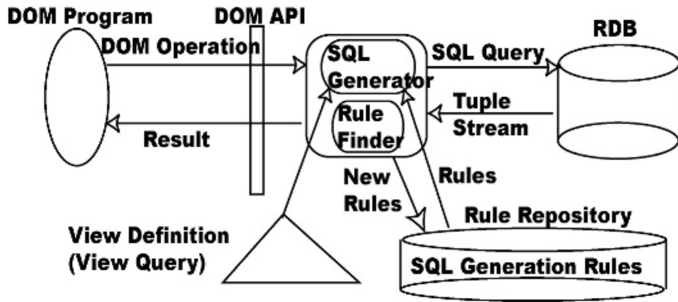


Fig. 1. Architecture

the execution, the rule generator tries to find SQL generation rules for efficient evaluation and stores them in the rule repository. The SQL generation rules are used in later executions.

An interesting issue in the approach is how to find SQL generation rules. A feature of our proposed method is that it provides a simple and efficient algorithm to find SQL generation rules. It is efficient both in time and required spaces.

The contributions of the paper are as follows: (1) We propose a general framework for applying XML application programs using DOM API to existing relational databases. (2) We explain an efficient algorithm to find SQL generation rules that are used for efficient executions of the DOM programs. (3) We give the results of our preliminary experiments, which show that the SQL generation rules can dramatically reduce execution costs.

Related Work. SilkRoute [2] and Xperanto [5] are well-known systems that publish XML data from relational databases. They provide mechanisms to construct *XML views* over relations and allow declarative XQuery-style queries against the views. Our system adopts the same mechanism as the SilkRoute's to define XML views, but allows any procedural program using DOM operations to access the XML views. The difference is essential and requires new optimization techniques.

To the best of our knowledge, ROLEX [1] is the only system that realizes DOM operations of XML views of relational data. ROLEX is different from our system in that it is designed to use a main-memory database system. The focus is mainly on utilization of data management facilities, such as concurrency control and recovery features, of the main-memory database system. In contrast, our system is designed to give an easy way to apply latest programs using XML-technologies to ubiquitous, possibly legacy data stored in relational databases. For optimization, ROLEX introduces a navigational profile to represent the probability of the application navigating along edges in the DOM tree. Our SQL generation rules provide a finer granularity control of SQL generation, which is driven by the status of running DOM programs.

DBDOM [7] is a persistent DOM implementation that uses RDBMs to store XML data. DBDOM defines a fixed database schema and cannot work as a bridge between DOM programs and existing relational data. So our system and DBDOM have completely different purposes.

2 Motivating Example

We motivate the problem of generating efficient SQL queries for DOM operations with an example. We use TPC-H benchmark database [6], which contains information on parts, the suppliers of those parts, customers, and their orders. Figure 2 shows a part of its database schema.

Supplier(suppkey, name, addr, nationkey)
Nation(nationkey, name, regionkey)
Region(regionkey, name)
PartSupp(partkey, suppkey, availqty)
Part(partkey, name, brand, size)

Fig. 2. Part of TPC-H database schema

XML Views of Relational Data. We assume that the information in the relational database is exported in the XML format determined by the schema in Figure 3. Each `supplier` element contains a `suppinfo` and a list of the supplier's `parts`. Each `suppinfo` contains the supplier's `name`, its `nation`, and the `region` to which it belongs. Each `part` element contains its `name`.

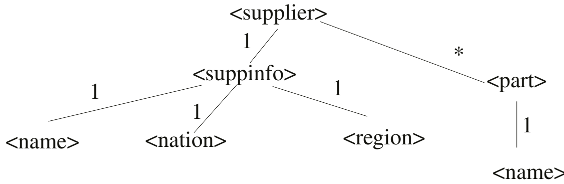


Fig. 3. Schema of XML view over the database

We use RXL [2] to define the XML view. Figure 4 shows the RXL view query mapping the relational data to XML data conforming to the schema in Figure 3. RXL's semantics is simple: As in SQL, *tuple variables* specified in the `from` clauses iterate over tuples of tables. For example, tuple variable `$s` iterates over the `Supplier` table. The `where` clause has conditions over these variables. For example, `$s.nationkey` = `$n.nationkey` is a join condition. The `construct` clause specifies an XML fragment to output. If a binding to the tuple variables specified in the `for` clause satisfies the condition in the `where` clause, the fragment in the `construct` clause is generated. The `construct` clause can

```
from Supplier $s
construct
  <supplier> <suppinfo><name>$s.name</name>
    { from Nation $n
      where $s.nationkey = $n.nationkey
      construct
        <nation>$n.name</nation>
        { from Region $r
          where $n.regionkey = $r.regionkey
          construct <region>$r.name</region> }
    }</suppinfo>
    { from PartSupp $ps, Part $p
      where $s.suppkey = $ps.suppkey,
            $ps.partkey = $p.partkey
      construct
        <part> <name>$p.name</name> </part> }
  </supplier>
```

Fig. 4. RXL view of TPC-H database

contain nested sub-queries surrounded by block boundaries "{" and "}". For example, there are three sub-queries in Figure 4.

Given an XML element specified in a `construct` clause, it is easy to construct an SQL query that computes instances of the given element; An XML element in a `construct` clause is generated for each tuple of the result of *joins* of relations in its `from` clause and that of superqueries. For example, instances of `<name>` and `<nation>` elements are generated for each tuple of the following queries' results:

```
select $s.suppkey, $s.name
from Supplier $s

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

This is best summarized by a tree structure called *viewtree* [2] (Figure 5). In the viewtree, we associate to each node a unique identifier based on Dewey Decimal Encoding[3]. For example, nodes with identifiers `N1.`x are children of the node `N1`. Each viewtree node is also associated with a *query* to compute instances of the XML element represented by the viewtree node. For example, $q(N1.1.2)$ specifies how to compute instances of `<nation>` element represented by `N1.1.2`; For each tuple in the result of `Supplier $s` $\bowtie$ `Nation $n`[1], one `<nation>` instance is generated. *Key attributes* to identify each such tuple (e.g. `$s.suppkey`) is annotated to each viewtree node. Some of nodes have additional attributes for element values (e.g. `$s.name`). In the following, we call an XML element instance an *XML node*.

Formally, each (computed) XML node is represented by pair $(n, \bar{k})$, where n is a viewtree node identifier and $\bar{k}$ is a sequence of values of the key attributes as-

[1] We omit join conditions for simplicity.

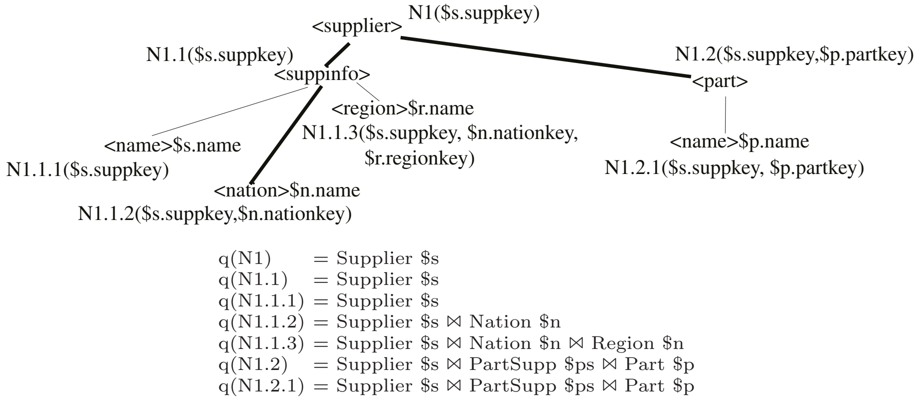


Fig. 5. Viewtree $\mathcal{T}_1$ (Bold lines represent $\mathcal{T}_1'$)

sociated with viewtree node n. The tag associated to XML node $(n, \bar{k})$ is the tag associated to n in the viewtree. The parent-child relationship among XML nodes are defined as follows: $(n_1, \bar{k}_1)$ is a child of $(n_0, \bar{k}_0)$ if n_1 is a child of n_0 in the viewtree, and the values in $\bar{k}_1$ has the same values for the same attributes in $\bar{k}_0$. For example, XML node `(N1.1.2, [$s.suppkey=#s1, $n.nationkey=#UK])` is a child of `(N1.1, [$s.suppkey=#s1])`.

DOM Operations on XML Views. The DOM (Document Object Model) is a programming API for documents [8]. Intuitively, it models a document instance as an object (node) tree that reflects the structure of the document. We use the program in Figure 6 (Ignore superscripts for a while) as a running example. The program takes as input an XML document, which is given by file name v (Line 2), traverses the document's element hierarchy in the depth-first order (Line 4, Lines 7-14), and outputs each element's tag name and content (Line 10-11). Typically, a DOM operation takes an XML node and returns one or more XML nodes. In the program, `getFirstChild` method (Line 8) returns the first child of the given XML node. Assume that the method is applied to an XML node `(N1.1, [suppkey=#s2])`. Then, the SQL query to compute the result can be obtained by adding selection condition `$s.suppkey=#s2` to the query for the viewtree node `N1.1.1`:

```
select $s.suppkey, $s.name
from Supplier $s
where $s.suppkey=#s2
```

Another example is the `getNextSibling` method (Line 9), which returns the next sibling in the tree structure of XML instance nodes. Assume that the result of the previous `getFirstChild` method is `(N1.1.1, [suppkey=#s2])` and we apply the `getNextSibling` method to the result. Since the viewtree node identifier of the current XML node is `N1.1.1`, we need the query of `N1.1.2`:

```
 1. void main() {
 2.    Document doc= new Document(v) (id1);
 3.    Node root=doc.getDocumentElement()(id2);
 4.    show(root)
 5. }
 6.
 7. void show(Node node) {
 8.    for (Node n=node.getFirstChild()(id3); n!=null;
 9.                n=n.getNextSibling()(id4)) {
10.      System.out.println(n.getNodeName()(id5));
11.      System.out.println(n.getNodevalue()(id6));
12.      show(n);
13.    }
14. }
```

Fig. 6. Fragment of DOM program $\mathcal{P}_1$

```
select $s.suppkey, $n.nationkey, $n.name
from Supplier $s, Nation $n
where $s.nationkey=$n.nationkey
      and $s.suppkey=#s2
```

Assume that we apply the `getNextSibling` method to `(N1.2, [suppkey=#s2, partkey=#p3])`. In this case, the siblings are computed by the same query, since one `<supplier>` may have more `<part>`s (See Figure 3). The system submits the following query:

```
select $s.suppkey, $s.partkey,
from Supplier $s, $PartSupp $ps, Part $p
where $s.suppkey=$ps.suppkey,
      $ps.partkey=$p.partkey,
      $s.suppkey=#s2
order by $suppkey, $s.partkey
```

and retrieves the next tuple of the tuple satisfying the condition `$s.partkey=#p3`. Note that the `order by` clause is used to enforce a fixed order among siblings; The siblings are sorted by key attributes.

Generating SQL queries in this way is simple but inefficient, since the system has to submit the same number of SQL queries as DOM operations. We call the SQL generation schema above the *naive SQL generation*.

2.1 Sorted Outer Union Plans and Introduction of ADO Labels

But what if the execution of DOM operations follows a particular order? A simple example is a program that traverses XML nodes in the depth-first order. In this case, *sorted outer-union* plans [5] are effective. A sorted outer-union plan is a particular form of SQL query to compute XML query results on XML views of relational databases. Intuitively, the query result is a relation that corresponds to an *unnested* form of the result XML document, where each tuple corresponds to one instance of XML element or path. We use a variation of sorted outer-union plans here.

```
 1. select * from (
 2. select 1 as L1, $s.suppkey, N as L2, N, N as L3, N, N                 // N1
 3. from Supplier $s
 4. union
 5. select 1 as L1, $s.suppkey, 1 as L2, N, N as L3, N, N                 // N1.1
 6. from Supplier $s
 7. union
 8. select 1 as L1, $s.suppkey, 1 as L2, N, 2 as L3, $n.nationkey, $n.name // N1.1.2
 9. from Supplier $s, Nation $n
10. where $s.nationkey=$n.nationkey
11. union
12. select 1 as L1, $s.suppkey, 2 as L2, $p.partkey, N as L3, N, N        // N1.2
13. from Supplier $s, $PartSupp $ps, Part $p
14. where $s.suppkey=$ps.suppkey, $ps.partkey=$p.partkey,
15. )
16. order by L1, $suppkey, L2, $p.partkey, L3, $n.nationkey
```

Fig. 7. Sorted outer union plan for $\mathcal{T}_1'$

To explain our sorted outer-union plans, we use XML view $\mathcal{T}_1'$ that is a subtree of XML view $\mathcal{T}_1$. In Figure 5, $\mathcal{T}_1'$ is represented by the bold lines. Figure 7 is an outer-union plan SQL query for a depth-first traversal of the XML view. Here, `N` means a null value. The query is a naive implementation of the outer-union plan. More sophisticated implementations are given in [5] [2]. In the result of the SQL query, each tuple corresponds to one instance of an XML element. In each tuple, XML element instance $(n, \bar{k})$ is encoded as follows: (1) Viewtree node identifier $n = \mathtt{N}\ l_1.l_2.\ldots l_n$ is decomposed and stored in the attributes $L1$, $L2$, ..., and Ln. (2) key values in $\bar{k}$ are stored in their corresponding attributes.

The `order by` clause sorts the tuples by $L1$, key attributes for the root node in the viewtree (i.e., `$s.suppkey`. We use $k_{(1,i)}$ to denote the attributes), $L2$, key attributes that appear for the first time in the viewtree nodes at level 2 (i.e., `$p.partkey`. $k_{(2,i)}$), etc. The tuple order is compatible with the the depth-first traversal of the XML view of the relational data (Figure 8), and traversal in the depth-first order is achieved by executing the query and reading the tuples in a sequential way. In other words, if we assign to each node label "$l_1, k_{(1,1)}, \ldots, k_{(1,n_1)}, l_2, k_{(2,1)}, \ldots, k_{(2,n_2)}, l_3, k_{(3,1)}, \ldots,$", and define the *lexicographic order* on the labels, the labels *encodes* structural relationship among XML nodes. We call the framework the *Augmented Dewey Order Encoding* here and name each such label an *ADO label*. Note that given an ADO label for an XML node, we can easily extract the viewtree node identifier $l_1.\ldots.l_n$ whose query computed the XML node. As explained later, our proposed method utilizes the characteristic of the encoding framework.

The sorted outer-union plan is much more efficient compared to the naive SQL generation for the program $\mathcal{P}_1$: The query is executed only once, and the tuples in the query result all contribute to the DOM operations' results. In contrast, as explained in Section 2, these are not true in the naive SQL generation. In general, sorted outer union plans are effective when the program is something like the depth-first traversal (details are explained in Section 3).

2.2 Problem and Our Approach

It is not a trivial task, however, to extract such "effective" patterns from the program source code. There are so many different ways to implement the same

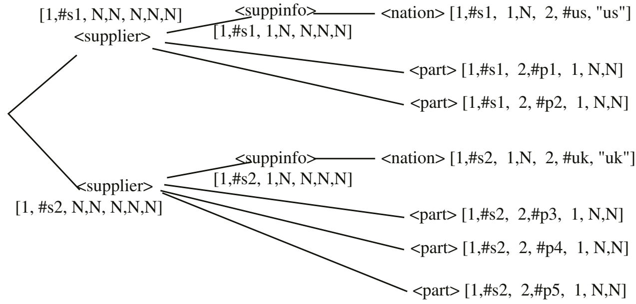


Fig. 8. Structure of an XML instance and ADO labels

```
 1. void show(Node node) {
 2.   if (node!=null) {
 3.      System.out.println(node.getNodeName());
 4.      System.out.println(node.getNodevalue());
 5.   }
 6.   Node n1=node.getFirstChild();
 7.   show(n1);
 8.   Node n2 node=n1.getNextSibling();
 9.   show(n2);
10.   }
11. }
```

Fig. 9. Another DOM program

functions in programs. For example, the program in Figure 6 and the one in Figure 9 output the same results if their input XML data is a binary tree. In the extreme case, a program may *hard-wire* the complete structure of XML instance. In addition, it is not always true that a program has a *perfect* pattern like the depth-first-order example above. For example, a program that traverses XML data in the depth-first order but skips over a small number of XML nodes, should be efficiently supported by the same outer-union plan query. Considering these, we choose not to analyze the program source code, nor to match predefined patterns with the code. Instead, we developed a technique that observes program executions to find better SQL generation rules. The rules found are used for later executions of the program with different data. This is explained in detail in the following section.

3 Optimization Technique

3.1 How the Rule Finder Works

The rule finder (Figure 1) takes as input a stream of DOM operation results. Formally, it takes pairs $(\langle id, a\rangle, l)$ where id is an *operation identifier* associated with each DOM operation in the program (See Figure 6) and a is a collection of *parameter values* of the operation. l is an ADO label (i.e., a tuple representing an XML node instance) that is the result of the operation's execution. To make it possible for the system to take ADO labels even in the first execution, attributes for viewtree node identifiers, such as `L1` and `L2`, are incorporated into SQL queries for the naive SQL generation.

Each such pair is given to the rule finder every time when an operation having *id* with parameter values a is executed and the returned answer is an XML node whose ADO label is l. The stream is used to find SQL generation rules.

In later executions of the program, the SQL generator uses the stored rules to make more efficient SQL queries having the same effects with the program and the XML view definition. The rule finder keeps trying to find other new rules because one execution of a program does not necessarily go through every possible execution patterns of the program. A key issue in the architecture is how to find SQL generation rule quickly so that the system can take advantages of the latest findings as soon as possible.

3.2 SQL Generation Rules

Let $\mathcal{T}$ be a given viewtree and $\mathcal{P}$ be a prgram. An SQL Generation Rule (shortly, a *rule*) over $\mathcal{T}$ and $\mathcal{P}$ has the form of $p \to t$. Here, p is a sequence of pairs $\langle id_i, a_i \rangle$ where id_i is an *operation identifier* and a_i is a collection of *parameter values* of the operation. In a_i, obj is used as a special parameter whose value is the object to which a method is applied.

In each rule, t is either $\mathcal{T}$ or a contraction of $\mathcal{T}$. A contraction of a tree is a result of contracting the tree by removing some of the nodes [4]. Unlike subtrees, a contraction can contain nodes that are not adjacent to each other in the original tree. The following (R_1) is a rule over $\mathcal{T}_1$ in Figure 5 and $\mathcal{P}_1$ in Figure 6:

$$R_1 : [\langle id1, \phi \rangle, \langle id2, [obj = \texttt{N1}] \rangle] \to \mathcal{T}_1.$$

R_1 says that if operation $id2$ (Line 3, Figure 6) is applied to an XML node having viewtree node identifier N1, following an execution of operation $id1$ (Line 2), an outer-union plan SQL query for $\mathcal{T}_1$, similar to the one in Figure 7, should be generated and executed.

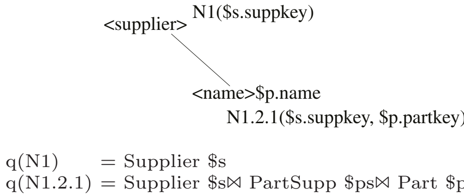
Note that parameter values in a_i do not contain XML node instance $(n, \bar{k})$ but contain viewtree node identifier n (e.g., N1) only. This is because a pattern containing particular XML nodes is too strong in the sense that it cannot match the data even if values in the relational data are slightly changed.

The rule above is a special case where parameter values for methods contain no value other than XML node and $\mathcal{T}$ is the original viewtree itself. Let $\mathcal{P}_4$ be a DOM program[2] that contains operations with $id8$ and $id9$ and the latter operation is `a.getElementsByTagName(x)`. The following is a more general example of a rule over $\mathcal{T}_1$ and program $\mathcal{P}_4$:

$$R_2 : [\langle id8, [obj = \texttt{N1.2}] \rangle, \langle id9, [obj = \texttt{N1}, par1 = \text{"part"}] \rangle] \to \mathcal{T}_2$$

where $\mathcal{T}_2$ (Figure 10) is a contraction of $\mathcal{T}_1$ and `"part"` is a value of `x` observed when the system executes the operation with *id9*. The rule says that if the pattern matches the stream observed in the execution, an outer-union plan SQL query for $\mathcal{T}_2$ should be generated and executed.

[2] The entire code of $\mathcal{P}_4$ is omitted here.



q(N1) = Supplier $s
q(N1.2.1) = Supplier $s⋈ PartSupp $ps⋈ Part $p

Fig. 10. Contraction $\mathcal{T}_2$ of $\mathcal{T}_1$

3.3 Using Generation Rules

If the system knows an SQL generation rule $p \rightarrow t$ where $p = [\langle id_1, a_1\rangle, \langle id_2, a_2\rangle]$, the rule is used in the following way: First, the system watches an execution of the DOM program and takes as input the stream of pairs $(\langle id, a\rangle, l)$ from the execution. If p matches a subsequence of the stream, the system generates an outer union plan SQL query q for t, and adds selection conditions to q to avoid computing too many unnecessary tuples according to the types of operations. For example, if the operation id_2 is `getFirstChild`, conditions are added as explained in Section 2.

Then, the query q is submitted to the RDBMS, and subsequent DOM operations consume the tuples from its result, without submitting other SQL queries. How to choose necessary tuples from the result depends on types of DOM operations. For example, `getFirstChild` operation selects the tuples having ADO Labels compatible with the ADO label of its parent. This is done in a similar way as explained in Section 2. If the system cannot find necessary tuples in the result stream, the system returns to its normal job; it submits one SQL query to the RDBMS for each DOM operation while searching for subsequences of the stream that matches rules.

3.4 Observing Executions to Find Rules

As mentioned, a point of our method is that it gives a simple and efficient mechanism to find rules. The basic idea is to use ADO labels to (1) search the tuple stream for sequences of XML nodes that can be computed by outer-union plans and (2) to efficiently construct SQL generation rules. Compatibility with outer-union plans can be easily decided by checking if ADO labels of successive XML nodes follow the lexicographic order defined on ADO labels. Efficient construction of rules is realized by utilizing the fact that we can easily extract from ADO labels the information on the viewtree nodes who computed XML nodes.

Figure 11 is an algorithm to find rules each of which has a sequence p of length two. In other words, each rule generated by the algorithm has the form $[\langle id_1, a_1\rangle, \langle id_2, a_2\rangle] \rightarrow t$, like R_1 and R_2. Its input is a stream of pairs $(\langle id, a\rangle, l)$ (Line 4). The output is a set of rules, which is generated in turn (Line 15).

In the algorithm's execution, two successive pairs are kept in $(\langle id', a'\rangle, l')$ (Line 19) and $(\langle id, a\rangle, l)$. Variable t (Lines 1, 8, 11, and 15) represents a contraction of $\mathcal{T}$ for a rule. In t, each contraction is encoded as a set of viewtree nodes;

```
 1. t={};
 2. inOrder=false; // true when the input follows the order
 3. <id', a'>=<nil, []>; l'=infinite;
 4. for each (<id, a>, l) in the Input Stream {
 5.    if (l' < l) {
 6.       if (!inOrder) {
 7.          inOrder=true;
 8.          t.add(l'.getViewTreeNodeID());
 9.          <id_2, a_2>=<id', a'>;
10.       }
11.       t.add(l.getViewTreeNodeID());
12.    } else {
13.       if (inOrder) {
14.          inOrder=false;
15.          outputRule( [<id_1, a_1>, <id_2,a_2>]->t );
16.          t={};
17.       } else <id_1, a_1>=<id', a'>;
18.    }
19.    <id', a'>=<id, a>; l'=l;
20. }
```

Fig. 11. Algorihtm for finding rules

For example, $t = \{\texttt{N1}, \texttt{N1.2.1}\}$ represents a contraction with two nodes `N1` and `N1.2.1` (Figure 10).

Basically, what the algorithm does is just to compare ADO labels of two successive XML nodes (DOM operation results) (Line 5), and as long as the labels follow the lexicographic order, the algorithm adds viewtree nodes to t (Lines 8 and 11). The initial values of $(\langle id', a' \rangle, l')$ is $(\langle nil, [] \rangle, infinite)$ (Line 3), where $infinite$ is a special label that is greater than any other labels. If the algorithm finds a label on the stream not following the lexicographical order, it decides that the remaining DOM operation results should be computed by other SQL queries, outputs the rule constructed so far (Line 15), and resets t for the next rule (line 16).

Note that some DOM operations, such as `getElementsByTagName()`, return a list of XML nodes. For such operations, we use a special value `0` for l, where comparison predicates (like $<$) to compare `0` and any other value always return true. In other words, the algorithm ignores the nodes in the list at that time. But the nodes affect the rule generation process anyway, since sooner or later each such node should be extracted from the list, by using other methods like `item()`.

The algorithm is quite efficient with time complexity $O(S)$ and space complexity $O(V)$, where S is the size of the input stream and V is the size of the viewtree.

4 Preliminary Experiments

This section shows the results of our preliminary experiments. The purpose is to examine the impact of using outer-union plan queries to process DOM operations.

The experiments were run using the following configuration: We use the TPC-H benchmark databases (with different scale factors) managed by the Post-

greSQL 7.3.2. The XML view query used is the one shown in Figure 5. The database server has Pentium-4 1.5GHz CPU and 512MB memory. The operating system is Linux RH 8.0. A DOM-compliant library that implements our proposed algorithm was run on the same database server.

First, we executed program $\mathcal{P}_1$ (Figure 6) on the XML view with different sizes of databases. This is an ideal situation where all the DOM operations constitute a depth-first traversal of the entire XML data. One outer-union plan SQL query is generated. The result is shown in Figure 12 (left). The horizontal axis is the number of XML nodes contained by the XML view of each entire database. $\mathcal{P}_1$ generates all of the XML nodes in the depth-first order. The dotted line represents the execution time after SQL generation rules are found. As we can seen, the outer-union plan *dramatically* improves the performance.

Next, we executed a more realistic program $\mathcal{P}_3$ that retrieves some XML nodes from the XML view and traverses subelements of each retrieved XML nodes in turn (Figure 13). This is a typical fragment of a DOM program we can see in practical situations. Interestingly, the system generates only one SQL query, corresponding to viewtree nodes `N1.1`, `N1.1.1`, `N1.1.2`, and `N1.1.3` in Figure 5. This is because the result tuple stream is consistent with the lexicographic order defined on ADO labels. The result is shown in Figure 12 (right). $\mathcal{P}_3$ is faster than $\mathcal{P}_1$ since it retrieves and traverses a limited number of XML nodes. The size of database having 1,000,000 (virtual) XML nodes is about 405MB.

Finally, to show that our approach is scalable, we applied the optimized $\mathcal{P}_3$ to larger databases with the TPC-H scale factors 0.5 and 1, which means the sizes of databases are about 500MB and 1GB, respectively. The result is shown in Figure 14. We found that after SQL generation rules are obtained, both $\mathcal{P}_1$ and $\mathcal{P}_3$ run in linear time. In addition, as Figure 14 shows, $\mathcal{P}_3$ with SQL generation rules takes only 3.3 sec. for query evaluation even with 1GB database. It is possible that we can make the total execution time further smaller by considering other factors than SQL query evaluation, but it is beyond the scope of the paper.

Note that our approach requires only a *small and constant memory space*, even if application programs traverse the entire XML data. Applying DOM operations to (virtual) XML views is effective for large database since the system materializes only a required part of the data on demand. In contrast, usual DOM libraries would not work with a huge XML data since it requires the entire XML data to be materialized at a time.

5 Conclusion and Future Work

This paper proposed a system that allows XML application programs using the DOM API to take as input existing relational data. The proposed system has a practical impact because a huge amount of data still remains in relational databases, while more and more XML application programs are being developed. The key issue is an *optimization framework* for DOM operations, interleaved with ordinary programming language codes, to deal with XML views of relational data. Our proposed optimization framework has a training mechanism

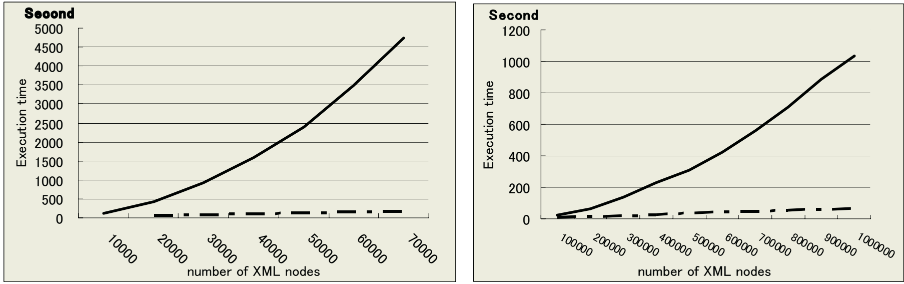


Fig. 12. Experimental results

```
1. void main() {
2.   Document doc= new Document(v);
3.   NodeList a=doc.getElementsByTagName("suppinfo");
4.   Node b=null;
5.   for (int i=0; (b=a.item(i))!=null; i++) { show(b); }
6. }
```

Fig. 13. The main program of $\mathcal{P}_3$ (`show` function is given in Figure 6)

Size	Total Time(sec)	Query Execution Time(sec)
500MB	77.4	1.7
1GB	154.7	3.3

Fig. 14. Results for large databases

to find SQL generation rules for efficient executions of the programs. We developed an algorithm to efficiently find such SQL generation rules. Our preliminary experiments showed that our approach is promising and scalable.

Future work includes development of algorithms to find more sophisticated SQL generation rules. We believe incremental evolution of rules is possible when we apply machine learning techniques to our approach. Another interesting issue is development of a *just-in-time* optimization mechanism to allow SQL generation rules to be applied during the same program execution as soon as they are found. Also, supporting updates is important, although the view update problem is known to be difficult in general.

Acknowledgments

We would like to thank Prof. Seiichi Komiya of Shibaura Institute of Technology for discussions, and we thank the reviewers for their detailed comments and suggestions. This research was partially supported by the Ministry of Education, Culture, Sports, Science, and Technology, Grant-in-Aid for Young Scientists (B) (15700108).

References

1. P. Bohannon, S. Ganguly, H. F. Korth, P. P. S. Narayan, P. Shenoy: Optimizing View Queries in ROLEX to Support Navigable Result Trees. Proc. VLDB 2002, 119-130, 2002.
2. M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, WC Tan. SilkRoute: A Framework for Publishing Relational Data in XML. ACM Trans. Database Syst. 27(4): 438-493, 2002.
3. Online Computer Library Center. Introduction to Dewey Decimal Classification. `http://www.oclc.org/oclc/fp/about/about_the_ddc.htm`
4. M. Atallah (ed). Algorithms and Theory of Computation Handbook, CRC Press, 1998.
5. J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, B. Reinwald: Efficiently publishing relational data as XML documents. VLDB Journal 10(2-3): 133-154, 2001.
6. Transaction Processing Performance Council. TPC-H (Decision Support for Ad Hoc Queries) `http://www.tpc.org`
7. DBDOM Home Page. `http://dbdom.sourceforge.net`
8. W3C. Document Object Model (DOM). `http://www.w3.org/DOM/`