Processing XML View Queries Including User-defined Foreign Functions on Relational Databases

Yoshiharu Ishikawa† Jun Kawada‡ Hiroyuki Kitagawa†
† Institute of Information Science and Electronics
‡ Graduate School in Systems and Information Engineering
University of Tsukuba
{ishikawa,kitagawa}@is.tsukuba.ac.jp

Abstract

With the increased popularity of XML, XML publishing of RDBs has been attracting a lot of research interests. One of typical approaches is to use a middleware system to render XML views over RDBs and to allow users to access data with XML query languages such as XQuery. The query processing is done efficiently by making the best of the querying power of RDBMSs. Namely, XML queries are translated into SQL queries and tagging operations, which are processed by the RDBMSs and middleware, respectively. In some XML query languages including XQuery, use of user-defined foreign functions is enabled or planned as an extension feature to cope with domain dependent semantics. Foreign functions are defined for XML fragments, and their implementations are often given by codes in a general programming language. The existing query processing schemes on XML views do not consider cases where foreign functions are included in XML queries. In this paper, we propose extended schemes to process XML queries in such cases. In the proposed schemes, the middleware takes care of processing foreign functions as well as tagging operations. Therefore, the proposed schemes are applicable to XML views on commonly available RDBMSs. Three types of query processing schemes are proposed, and their performance is studied with experiments.

1. Introduction

XML (Extensible Markup Language) has become a standard format to exchange information over the Internet, and the importance of database technologies that support storage, processing, and delivery of XML is still increasing [1, 6]. Since most of data for enterprise computing is stored in conventional relational databases (RDBs), XML publishing for selecting specified data items and transforming them to appropriate XML formats for delivery and publication is one of the important research topics.

As an approach to support XML publishing with database systems, it is proposed to construct virtual XML views over RDBs using middleware technologies and provide query facilities to such views for flexible and efficient generation of XML documents [3, 5, 4, 7, 8]. Figure 1 shows such an example of XML view for relational tables shown in Fig. 2.

The XPERANTO group of IBM Almaden has been developing a query processing system for XML views built over RDBs [7]. On receiving an XML query written in the XML query language XQuery [2] as in Fig. 3, the XPERANTO system generates an optimized SQL query to extract required tuples from the underlying RDBMS.

We should note, however, that some XML query languages including XQuery allow or try to incorporate features of definition and usage of user-defined foreign functions [2, 9, 10]. Figure 3 shows an example of a query including a user-defined foreign function isWider(). A foreign function is usually coded with a general-purpose programming language and assumes in-memory representation of the XML document fragment to which the function will be applied. Therefore, we need special consideration to process XML queries including user-defined foreign functions in evaluating XML view queries. To our knowledge, there are no previous proposals on query processing techniques for XML views with the support of user-defined foreign functions.

In this paper, we propose query processing methods for XML views over RDBs. Our approach assumes that a middleware system cooperates with the underlying RDBMS that has conventional SQL processing facility. For this purpose, we extend the XPERANTO approach considering foreign function processing. We propose three query processing strategies and
Two-step Processing Method:
1. Issue an SQL query to the RDBMS to obtain relational tuples required for the evaluation of foreign functions.
2. Obtain relational tuples as the result of the query.
3. Generate XML fragments from the obtained tuples and perform the foreign function evaluation.
4. Issue another SQL query to obtain the relational tuples for the generation of the final query result; the SQL query contains an additional condition by incorporating the result of Step 3. The additional condition can filter out tuples that do not satisfy the foreign functions.
5. Obtain the relational tuples for the generation of the final result.
6. Generate the result XML documents.

As shown in Section 5, we can take two approaches to Step 4: their difference is only whether a temporal table is used or not in their query processing.

One-Step Processing Method:
1. Issue an SQL query to the RDBMS to obtain relational tuples required for the evaluation of foreign functions and for the generation of the result XML documents.
2. Obtain relational tuples as the result of the query.
3. Generate XML fragments and evaluate foreign functions.
4. Filter out unqualified tuples based on the evaluation result of Step 3 then generate the result XML documents.

The details of the above methods are described in Section 5. In the next section, we briefly introduce the XML view query processing of XPERANTO, which is the basis of our query processing approach.

The paper [7] addressed to this problem briefly describing their idea for the extended XPERANTO system in its future work part. The idea is to enhance the RDBMS with the support of XML fragment generation functions and foreign function evaluation facility, and evaluate queries with foreign functions inside of the RDBMS. Since this approach requires an RDBMS enhanced with special implementation for foreign function support, it is not applicable to query processing in conventional RDBMSs.

In contrast to this special implementation approach, we process XML view queries including foreign functions by the cooperation of a conventional RDBMS and a middleware system. Since foreign functions are evaluated in the middleware instead of an RDBMS, our approach does not require special extensions to the underlying RDBMS. We propose two kinds of query processing methods: the two-step processing method (Fig. 4) and the one-step processing method (Fig. 5).
3. The XPERANTO Approach

3.1. Overview

In the XPERANTO approach, a virtual default XML view is automatically created for a relational table in the underlying RDBMS. Figure 6 shows the default view for the relational table shown in Fig. 2. Users can use the XQuery query language [2] to specify queries to default XML views and to define higher views over them. For instance, the view definition in Fig. 7 written in XQuery defines an XML view shown in Fig. 1. Based on this approach, the underlying RDB is hidden from users so that users can uniformly manipulate XML databases using XQuery.

| 01:  | <db> |
| 02:  | </db> |
| 03:  | <city> |
| 04:  | </city> |
| 05:  | <row> |
| 06:  | </row> |
| 07:  | </city> |
| 08:  | <location> |
| 09:  | </location> |
| 10:  | <city> |
| 11:  | </city> |
| 12:  | <facility> |
| 13:  | </facility> |
| 14:  | </facility> |
| 15:  | </location> |
| 16:  | </location> |
| 17:  | <city> |
| 18:  | </city> |
| 19:  | <facility> |
| 20:  | </facility> |
| 21:  | <facility> |
| 22:  | </facility> |
| 23:  | <facility> |
| 24:  | <facility> |
| 25:  | </facility> |
| 26:  | </facility> |

Figure 6. Default XML view

XQuery query processing consists of the following phases:

(1) **Query Parsing**: The XQuery query is parsed and transformed into an internal XQGM representation.

(2) **Query Transformation**: SQL queries to be issued to the RDBMS and sequence(s) of tag operators to be executed in the middleware are generated from the internal representation.

We call the sequence of Phases (1) and (2) query planning. In the following, we focus on the method of query planning.

3.2. Query Parsing

In the query parsing phase, the query defining the target XML view and a query specified by the user for the XML view (both written in XQuery) are transformed into internal graph structures in XQGM (XML Query Graph Model). They are called the view definition XQGM graph and the user query XQGM graph, respectively.

Figure 8 shows an example of view definition XQGM graph transformed from the view definition in Fig. 7. An XQGM graph is a directed graph consisting of nodes each of which corresponds to an extended relational algebra operator shown in Table 1. In Fig. 8, each of the nodes numbered 1 through 11 corresponds to an extended relational operator. The solid arrows represent input/output relationships among them. It should be noted that node 10 represents a correlated join operator. The operator means that corresponding (correlated) location tuples and facility tuples are joined for each city. The dotted arrows represent such correlation relationships.

Figure 7. XQuery for the XML view definition

Figure 8. View definition XQGM graph

XQGM operators can call XML functions (e.g., tag operators to attach XML tags to relational tuples) inside of them. Table 2 shows such XML functions and their corresponding XQGM operators. Note that node 11 in Fig. 8 is an abbreviated representation of operations to attach XML tags to the tuples obtained in node 10; tag operations actually performed is a cascaded sequence of XML function calls shown in node 12.
Table 1. XQGM operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Functionality</th>
<th>Operator</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>specify a relational table</td>
<td>project</td>
<td>projection operator</td>
</tr>
<tr>
<td>select</td>
<td>selection operator</td>
<td>join</td>
<td>join operator</td>
</tr>
<tr>
<td>groupby</td>
<td>grouping operator</td>
<td>orderby</td>
<td>attribute-based ordering operator</td>
</tr>
<tr>
<td>unnest</td>
<td>returns an unnested list</td>
<td>union</td>
<td>union operator</td>
</tr>
</tbody>
</table>

Table 2. XML functions and their corresponding operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>cr8Elem(Tag, Atts, Clist)</td>
<td>create an element consists of an element name Tag, an attribute list Atts, and element content Clist</td>
<td>project</td>
</tr>
<tr>
<td>cr8AttList(Att1, ..., Attn)</td>
<td>create an attribute list consists of parameter attributes</td>
<td>project</td>
</tr>
<tr>
<td>cr8Att(Name, Val)</td>
<td>create an attribute with an attribute name Name and an attribute value Val</td>
<td>project</td>
</tr>
<tr>
<td>cr8XMLFragList(Clist)</td>
<td>create an XML fragment list from element content parameters such as elements and texts</td>
<td>project</td>
</tr>
<tr>
<td>aggXMLFragList(L)</td>
<td>aggregate functions to generate an XML fragment list from the input</td>
<td>groupby</td>
</tr>
<tr>
<td>getTagName(Elem)</td>
<td>returns the element name of an element Elem</td>
<td>project, select</td>
</tr>
<tr>
<td>getAttributes(Elem)</td>
<td>returns the attribute list of an element Elem</td>
<td>project, select</td>
</tr>
<tr>
<td>getContent(Elem)</td>
<td>returns the XML fragment list which is the contents of an element Elem</td>
<td>project, select</td>
</tr>
<tr>
<td>getAttName(Att)</td>
<td>returns the attribute name of an attribute Att</td>
<td>project, select</td>
</tr>
<tr>
<td>getAttValue(Att)</td>
<td>returns the attribute value of an attribute Att</td>
<td>project, select</td>
</tr>
<tr>
<td>isElement(Elem)</td>
<td>returns true if it is an element; otherwise returns false</td>
<td>project, select</td>
</tr>
<tr>
<td>isText(T)</td>
<td>returns true if T is a text; otherwise returns false</td>
<td>project, select</td>
</tr>
</tbody>
</table>

Table 3. Function composition rules

<table>
<thead>
<tr>
<th>Function</th>
<th>Composition Target</th>
<th>Composition Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTagName</td>
<td>cr8Elem(Tag, Atts, Clist)</td>
<td>Tag</td>
</tr>
<tr>
<td>getAttributes</td>
<td>cr8Elem(Tag, Atts, Clist)</td>
<td>Atts</td>
</tr>
<tr>
<td>getContent</td>
<td>cr8Elem(Tag, Atts, Clist)</td>
<td>Clist</td>
</tr>
<tr>
<td>getAttName</td>
<td>cr8Att(Name, Val)</td>
<td>Val</td>
</tr>
<tr>
<td>getAttValue</td>
<td>cr8Att(Name, Val)</td>
<td>Val</td>
</tr>
<tr>
<td>isElement</td>
<td>cr8Elem(Tag, Atts, Clist)</td>
<td>True</td>
</tr>
<tr>
<td>isElement except for cr8Elem</td>
<td>cr8Att(Name, Val)</td>
<td>False</td>
</tr>
<tr>
<td>isText</td>
<td>cr8XMLFragList(Clist)</td>
<td>True</td>
</tr>
<tr>
<td>isText except for PCDATA</td>
<td>cr8XMLFragList(Clist)</td>
<td>False</td>
</tr>
<tr>
<td>unnest</td>
<td>cr8XMLFragList(Clist)</td>
<td>unnest</td>
</tr>
<tr>
<td>unnest</td>
<td>cr8AttList(Att1, ..., Attn)</td>
<td>Att1</td>
</tr>
</tbody>
</table>

Figure 9 shows the user query XQGM graph for the user query shown in Fig. 3. In contrast to a view definition XQGM graph which represents the steps to derive an XML view from relational tables, a user query XQGM graph represents the steps to derive the query result from the XML view (represented by node 1 in Fig. 9). Three rectangles in Fig. 9 correspond to the for clause, where clause, and return clause, respectively, in the original user XQuery query. The edge from node 11 to node 18 in Fig. 9 represents that there is an existential qualification relationship between them. It means that only when the where clause represented by a subtree rooted at node 11 is true for some city, the joined result for the city is generated at node 18 and included in the subquery result.

As we describe in Section 4, we have extended the original XQGM proposed in XPERANTO to incorporate foreign function evaluation facility and to generalize its query processing approach. Actually, user query XQGM graphs as shown in Fig. 9 contain some extensions from the original ones.

3.3. Query Transformation

In this phase, we integrate the user query XQGM graph with the corresponding view definition XQGM graph then generate SQL queries and sequence(s) of XML tag operators. The outline of this step is as follows.

1. **View Composition**: In this step, the user query XQGM graph and the view definition XQGM graph are composed. For instance, node 1 in Fig. 9 is replaced with the subtree rooted at node 11 in Fig. 8 to form the composed XQGM graph. Next we compose XML functions for the generation of XML fragments. The XML functions come from the view definition XQGM graph and the user query XQGM graph, and we eliminate redundant XML fragment generation in the composition process. Table 3 shows the rules for the function composition. For example, getContents(\$cities) in node 2 in Fig. 9 and the toplevel function cr8XMLFragList(...) in node 12 of Fig. 8 are composed into the reduced form cr8XMLFragList(...). Function compositions are repeatedly applied to the composed graph to obtain its reduced representation.

2. **Computation Pushdown**: XQGM operators which can be processed within the RDBMS are pushed down towards the leaves of the XQGM graph as much as possible. The purpose of this step is to delegate the query processing role to the RDBMS to reduce the role of the middleware and to utilize the power of the RDBMS.

3. **Tagger Pull-Up**: Tag operators for the XML fragment generation are separated from other XQGM operators then moved upwards. Then SQL queries are created from the operators remaining in the leaf side of the graph.

4. Finally the SQL queries are integrated using the outer union operator for the generation of the result XML document structure. As proposed in [8], the integration uses the outer join operator to reduce the XML fragment generation cost and the required memory size in the middleware.
4. Extension to the XPERANTO Approach

4.1. Our Approach

In this section, we describe our extension to the original XPERANTO approach and present some assumptions to formulate our query processing methods. The paper of the original XPERANTO approach [7] only shows simple transformation examples and does not mention general cases; their approach has the following problems:

- Computation pushdown processing only considers top-level relations (e.g., city table in our example): The procedure for computation pushdown processing to sub-level relations (e.g., facility table) is not well described.
- They only consider a simple selection XQuery query that only contains a where condition: For example, the query shown in Fig. 3, which includes an output specification in the return clause, cannot be processed in the proposed method of [7].
- They only assume a simple where clause with one condition: Queries with multiple conditions as in Fig. 3 are not considered.

In addition, their approach does not consider foreign functions. They assume that all conditions in a where clause can be pushed down to the RDBMS. However, conditions based on foreign functions should be evaluated in the middleware; we cannot pushdown such conditions.

Therefore, we extend the XPERANTO approach to alleviate these problems. Actually, the user query XQGM graph shown in Fig. 9 incorporates the extension; the differences from the original XPERANTO approach are 1) subgraph generation considering the contents of the return clause and 2) the incorporation of the intersect operator to integrate multiple conditions in the where clause (node 11 in Fig. 9). In the following Subsection 4.2, we describe the query transformation method based on this extension.

Next we show our assumptions with respect to XML views, view definition and user XQuery queries, and foreign functions:

- Assumptions on XML views: We can determine the tuple(s) required for the construction of an XML document element using key value(s). For example, the tuple required for the construction of each city element can be determined using cid, the key attribute of city table.

- Assumptions on view definition and user XQuery queries:
  - The for clause can only contain a path expression and the path expression does not include filtering conditions.
  - Multiple conditions in the where clause are concatenated with logical AND operators.
  - The return clause does not include a subquery.

- Assumptions on foreign functions:
  - A user query can contain a single foreign function in the where clause.
  - A foreign function can accept document elements, constants, and string literals as arguments.

Although the above restriction looks slightly strong in the light of XQuery's powerful query syntax, the restriction has been much more relaxed than the XPERANTO approach in [7], as explained above.

4.2. Query Translation

In the following, we describe our query transformation method extending the XPERANTO approach shown in Subsection 3.3. As described in Section 4.1, our extension mainly consists of the following three points:

- Inclusion of a foreign function in the where clause
- Handling output specifications in return clauses: For this purpose, we have to transform two paths coming from the where clause and the return clause independently.
- Treatment of multiple conditions in the where clause and computation pushdown to subrelations.

4.2.1. Decorrelation

First we describe the first step of the query transformation called decorrelation. As shown in Fig. 8 and Fig. 9, a target XQGM graph generally contains a correlated join operator to represent correlation relationships between the main query and its subqueries. Since the direct execution cost of a correlated join is quite large, it is required to transform a given XQGM graph to eliminate correlation relationships [8]. Although the original XPERANTO approach performs the decorrelation process after the composition of a user query XQGM graph and a view definition XQGM graph, we process the decorrelation before the composition since that makes the transformation process clear.

After the decorrelation, the view definition XQGM graph shown in Fig. 8 is transformed as in Fig. 10. This is performed by transforming the correlated join operator in Fig. 8 by joining city table with location table and facility table, respectively. Note that, as shown in the figure, we need to incorporate left/right outer join operators to retain the semantics of the original query [7].

Similarly, we can perform decorrelation to the user query XQGM graph shown in Fig. 9. We append the subgraph rooted at node 4 (corresponds to the for clause) to each subgraph for the where clause and the return clause. Also, we translate the correlated join operator into left/right outer joins considering the existential conditions in the where clause. The result is shown in Fig. 11.

4.2.2. View Composition

Next, we perform view composition. After replacing nodes 1 and 12 in Fig. 11 with the content of Fig. 10, we compose XML functions for fragment generation and XML manipulation functions in the user query XQGM graph using the function composition rules shown in Table 3, then we get the XQGM graph shown in Fig. 12. Note that the where clause and the return clause form different paths due to our view composition approach.

4.2.3. Computation Pushdown

As described in Subsection 3.3, the next step is to pushdown XQGM operators towards the leaves of the graph to evaluate them in the RDBMS as much as possible. In contrast to
the original XPERANTO approach, since we assume that foreign functions are evaluated in the middleware instead of the RDBMS, we need to extend the pushdown approach of the XPERANTO in terms of our context. We assume that arguments of foreign functions are materialized as XML fragments inside of the middleware when we evaluate them. Therefore, we need to place the foreign function call operator to the point after the target XML fragment is generated using the project operator. In Fig. 12, for instance, since the evaluation of the foreign function \texttt{isWider()} in node 5 requires location information generated in node 4, we should not push the function call further down towards the leaf side.

### 4.2.4. Tagger Pull-Up

Finally, we perform tagger pull-up and get Fig. 13. First we transform nodes 1, 2, 3, 6, and 7 in Fig. 12 to nodes 1, 2, 3, and 7 of Fig. 13 by considering their semantics. Moreover, we transform node 8 in Fig. 12 to nodes 1 and 7 in Fig. 13 by propagating the constraint condition of the \texttt{where} condition (node 6 in Fig. 12) to the \texttt{where} clause. Next we replace XML functions such as \texttt{cr8Elem}, \texttt{cr8Att}, \texttt{cr8XMLFragList}, and \texttt{cr8AttList} with tag operators. Then an edge representing the correlation relationships are created by considering the left outer join semantics in nodes 13 and 14 of Fig. 12.

Tag operators \texttt{merge}, \texttt{aggregate}, and \texttt{input} in Fig. 13 represent query processing to be done in the middleware. The merge operator merges ordered input streams and implements XML functions \texttt{cr8Elem}, \texttt{cr8Att}, \texttt{cr8XMLFragList}, and \texttt{cr8AttList}. The aggregate operator performs ag-
5. Query Processing Architecture

5.1. Two-Step Processing Method

In the two-step processing method, we first issue an SQL query SQL-1 to the RDBMS and generate XML fragments for the foreign function evaluation from the retrieved tuples. Then the foreign function is evaluated. Next, we issue the query SQL-2 that incorporates the evaluation result of the foreign function to obtain the required relational tuples for the generation of the result XML document. Therefore, this approach is considered to be effective when the foreign function has a small selectivity factor. The flow of this method is shown in Fig. 14.

![Figure 13. XQGM after tagger pull-up](image)

![Figure 14. Flow of two-step processing method](image)

In the query planning part, the given user query is processed then SQL-1 and SQL-2 are generated. Also, XML tag operation sequences Tagger-1 and Tagger-2 are created; the former corresponds to nodes 4 and 5 in Fig. 13 and the latter does to nodes 11 to 16. They are used to generate the XML fragments for foreign function evaluation and the query result XML document, respectively.

The query execution part first obtains a tuple set Tuple-1 for the generation of the XML fragments from the RDBMS then sends it to the tagger. The tagger applies tag operations of Tagger-1 to Tuple-1 then generates a set of XML fragments Fragment for the foreign function evaluation. The foreign function evaluator evaluates the foreign function for Fragment then returns a set of qualified key values Keys (e.g., cid’s in our example) to the SQL query control part. The SQL query control part issues SQL-2 that includes filtering condition according to Keys to the RDBMS then obtains the result tuple set Tuple-2. Finally, the tagger applies tag operations of Tagger-2 to Tuple-2 and creates the result XML document.

With respect to the generation of SQL-2, we propose the following two approaches:

- **Two-Step Processing Method (where):** The qualified key value set Keys obtained by evaluating the foreign function is embedded into the where clause of SQL-2 as in "where fid in Keys".

- **Two-Step Processing Method (tmp):** Given the key value set Keys, this method first creates a one-column temporary table which contains key values of Keys. Then it incorporates a join operation with this temporary table in SQL-2.

We can utilize other methods such as a bitmap index-based approach, but here we take only the above two methods into consideration since they are easily implementable on commonly available RDBMSs.

5.2. One-Step Processing Method

The one-step processing method integrates two queries SQL-1 and SQL-2 in one SQL query, and obtains the relational tuples required for the evaluation of the foreign function and the result XML generation in one step. In this way, it can reduce the number of RDBMS invocations. It will be effective when the selectivity of the foreign function is relatively large. The flow in this method is shown in Fig. 15.
shown in Fig. 2 are constructed in the following settings: the tuple set Tuple in that we add a discrimination flag to each relational tuple in the tuple set Tuple from the RDBMS to specify whether the tuple is for the foreign function evaluation or for the result XML generation. The tagger generates a set of XML fragments Fragment from the tuple set Tuple for the evaluation of the foreign function. Finally, the middleware uses Keys, the result key value set of the foreign function evaluation, to select tuples that form the final query result, then the tagger generates the result XML document from these tuples.

In this section, we have described the proposed three query processing methods. Sample SQL queries to be issued to the RDBMS are shown in the next section.

### 6. Experimental Evaluation

In this section, we perform experimental evaluation for the three processing methods and compare there results.

#### 6.1. Outline of the Experiments

For the experiments, we use PostgreSQL (version 7.1) on a PC (Celeron 300MHz, memory 196MB) with Linux operating system. Three tables city, location, and facility shown in Fig. 2 are constructed in the following settings:

- No. of tuples in city table: $N = 1000$
- No. of tuples in location table: 10N and 100N
- No. of tuples in facility table: 10N and 100N

We construct an index on each cid attribute of city, location, and facility tables.

The following four queries are processed under different parameter sets.

- **Q1**: For each city whose area is larger than $X$, show its name and facilities.
- **Q2**: For each city whose area is larger than $X$, show its name, location information, and facilities.
- **Q3**: For each city whose area is larger than $X$ and whose population is larger than $Y$, show its name and facilities. This query corresponds to the example in Fig. 3 when $X = 10000$ and $Y = 10$.
- **Q4**: For each city whose area is larger than $X$ and whose population is larger than $Y$, show its name, location information, and facilities.

We assume that the condition “the area is larger than $X$” can be represented by the foreign function call `isWider()` and the condition “the population is larger than $Y$” can be represented by a conventional SQL condition. We vary selectivity factors for each condition as follows:

- Selectivity factors for the area condition (isWider()): $S_a = 0.1, 0.3, 0.5, 0.7, \text{ and } 0.9$
- Selectivity factors for the population condition: $S_p = 0.1$ and $0.3$

The query processing cost of the two-step processing method mainly consists of the following processing costs:

1. **processing cost for SQL-1**
2. **XML fragment generation cost for the foreign function evaluation**
3. **foreign function evaluation cost**
4. **processing cost for SQL-2**
5. **result XML generation cost**

The query cost of the one-step processing method consists of the costs below:

1. **processing cost for SQL-3**, the unified query created from SQL-1 and SQL-2 using the outer union operator
2. **XML fragment generation cost for the foreign function evaluation**
3. **foreign function evaluation cost**
4. **result XML generation cost**

The foreign function evaluation cost and the result XML generation cost are relatively small and both are included in the two processing methods. Also, the foreign function evaluation costs of the both processing methods are estimated to be equal. Therefore, we exclude these costs and compare the efficiency in terms of the remaining costs. Therefore, for the two-step processing method, we consider the gloss processing costs for SQL-1 and SQL-2. For the one-step processing method, we consider the processing cost for SQL-3.

#### 6.2. Sample SQL Queries

In this subsection, we show the SQL queries for query Q3 used in our experiment as an example.
6.2.1. Two-Step Processing Method (where)

Figure 16 shows the SQL query SQL-1 to retrieve tuples for the evaluation of the foreign function. It contains redundant representation since it is automatically obtained from the XQGM graph after tagger pull-up shown in Fig. 13. In lines 1 through 12, it first creates views to specify target relational tuples, which correspond to nodes 1, 2, 3, 7, 9, and 10 in Fig. 13. In lines 13 through 16, it defines a view to specify tuples to be retrieved, which corresponds to node 4 in Fig. 13. Lines 17 through 19 represent a query to be issued to the RDBMS. The roles of the outer UNION clause, target and type attributes, and the ORDER BY clause in line 19 are described later.

```
01: create view citytable as
02: select c.cid, c.cname
03: from city c
04: where c.population >= 10;
05: create view coordtable as
06: select c.cid, p.x, p.y
07: from citytable c, location p
08: where c.cid = p.cid;
09: create view faciltable as
10: select c.fid, f.fid
11: from citytable c, facility f
12: where c.cid = f.cid;
13: create view outerUnion1(target, type, cid, x, y) as
14: select INTEGER '0' as target, INTEGER '0' as type,
15: cid, x, y
16: from coordtable;
17: select *;
18: from outerUnion1
19: order by cid, target, type;
```

Figure 16. SQL-1 for two-step (where) processing method

As described in Subsection 5.1, foreign function evaluation is performed based on the tuple set obtained by the execution of SQL-1 and the evaluation result is saved in the key set Keys. SQL-2 shown in Fig. 17 is created based on Keys. The outer UNION clause in lines 1 through 9 sets city and facility tables required to generate the result XML document and corresponds to node 1 in Fig. 13. The view names citytable (line 5) and faciltable (line 9) are defined in SQL-1. In contrast to Fig. 16, it sets the flag value "1" to the target attribute; it means that this outer union tuple is used for the result XML generation (Note that the flags are used only in the one-step processing method). Each value of the type attribute is used to determine whether an outer union tuple comes from the city table or the facility table. The SQL query shown in lines 10 through 13 is issued to the RDBMS. The elements of Keys are inserted into the right hand of the IN operator (line 12), then filtering based on the result of the foreign function evaluation is performed. The role of ORDER BY clause in line 13 is to sort the outer union tuples; it can improve tag operation performance for the generation of the result XML document [8].

```
01: create tmp table keytable(cid text);
02: insert into keytable('keyval1');
03: insert into keytable('keyval2');
04: create unique index keytableindex on keytable(cid);
05: create view outerUnion2(target, type, cid, cname, fid, fname) as
06: -------
07: select INTEGER '1' as target, INTEGER '0' as type,
08: cid, null, null, fid, fname
09: from faciltable;
10: union all
11: select INTEGER '1' as target, INTEGER '0' as type,
12: cid, null, fid, fname
13: from faciltable;
14: union all
15: select INTEGER '1' as target, INTEGER '1' as type,
16: cid, null, fid, fname
17: from faciltable;
18: union all
19: order by cid, target, type;
```

Figure 17. SQL-2 for two-step (where) processing method

Although we have used views in defining queries SQL-1 and SQL-2, we can materialize them by using temporary tables and reuse the intermediate result of SQL-1. This idea can also be applied to the two-step processing method (tmp) described below. Concretely speaking, we can modify "create view coordtable" in Fig. 16 (line 5) to "create temp table coordtable". Similar modification can be applied to create view instatetable in line 9. We have performed some experiments in which we have replaced the view definitions with the temporary tables. We have found that the latter approach is not necessarily better than the former; their pros and cons depend on the experimental conditions. Therefore, we only report the results of the view definition cases due to the page limitation.

6.2.2. Two-Step Processing Method (tmp)

In the two-step processing method (tmp), SQL-1 is same as the one in the two-step processing method (where). Their difference is only on the treatment of the key value set Keys resulting from the foreign function evaluation. Fig. 18 shows SQL-2 for the two-step processing method (tmp). Lines 1 through 5 prepare for SQL-2; first a temporary table is created and the elements of Keys are inserted, then an index is constructed. The query shown in lines 17 through 19 performs a join operation with this temporal table and filters out the unqualified tuples.

```
01: create view outerUnion2(target, type, cid, cname, fid, fname) as
02: -------
03: select INTEGER '1' as target, INTEGER '0' as type,
04: cid, null, null, fid, fname
05: from faciltable;
06: union all
07: select INTEGER '1' as target, INTEGER '0' as type,
08: cid, null, fid, fname
09: from faciltable;
10: union all
11: select INTEGER '1' as target, INTEGER '1' as type,
12: cid, null, fid, fname
13: from faciltable;
14: union all
15: order by cid, target, type;
```

Figure 18. SQL-2 for two-step (tmp) processing method

6.2.3. One-Step Processing Method

As shown in Fig. 19, the one-step processing method issues a unified query SQL-3 obtained by combining SQL-1 and SQL-2 to the RDBMS. Each tuple with the attribute value 0 in the target attribute is selected from the result tuple set then an XML fragment is generated for the foreign function evaluation. The result of the foreign function evaluation is used to filter out the result tuples of SQL-3 with 1’s in the target attributes, then the tuples required for the generation of the result XML document are obtained.

6.3. Experimental Result

As described before, we perform experiments by fixing the number of tuples in the city table to be N = 1000. The number of tuples of the location and facility tables are set to 10N or 100N. In the following, we use a notation to specify a parameter condition; for example, we denote “(10N, 100N)” when the number of the facility tuples is 10N and the number of the location tuples is 100N.

6.3.1. Case of Q3 with $S_p = 0.3$

First, we show the result for query Q3 when the selectivity on the population attribute is $S_p = 0.3$ in Fig. 20. The
number of tuples of facility table is set to 10N and that of location table is set to 10V or 100N. The x-axis represents the selectivity values of the foreign function and the y-axis shows the wall-clock time required for the query processing. The figure says that the three methods have similar query costs irrelevant to the foreign function selectivities, but we can see that the one-step processing method is slightly better when (10N, 10N).

Next we show the result when we increase the number of facility tuples to 100N in Fig. 21. Since the facility information should be included in the result of Q3, the increase of the number of facility tuples means the increase of the volume of the query result. As shown in the figure, the two-step processing methods are effective when foreign function selectivities are small since filtering is well-performed. In particular, the two-step processing method (tmp) outperforms the one-step method in a broad range of parameters. Also, we should note that the two-step processing method (where) is worse than the two-step processing method (tmp) in both cases of (100N, 10N) and (100N, 100N). The two-step processing method (where) directly embeds the key value set into the condition part of SQL-2. It indicates that processing of SQL-2 takes much time when a large number of key values are given.

6.3.2. Case of Q3 with \(S_p = 0.1\)

In this case, the selectivity on the population attribute is small (\(S_p = 0.1\)). Fig. 22 shows the result for the cases of (10N, 10N) and (10N, 100N). In this case, the selection condition pushed down to the RDBMS works effectively to reduce the number of obtained tuples so that the one-step method is efficient in general.

We can observe the same tendency when we increase the number of facility tuples, namely for (100N, 10N) and (100N, 100N). In contrast to Fig. 21, the three methods result in almost similar performance since we can perform effective filtering based on the population condition, but the one-step method is slightly better than the others. The graph is omitted due to the space limitation.

6.3.3. Case of Q4 with \(S_p = 0.3\)

Next we show the result for Q4 with the selectivity on the population attribute is \(S_p = 0.3\). The difference of Q4 from Q3 is that Q4 outputs the location information for each qualified city in addition to the city name and facility information of Q3. Fig. 23 shows the result for the cases of (10N, 10N) and (10N, 100N). In the case of (10N, 100N), since the output size of the location information is large, the two-step
processing method, especially the two-step processing method (tmp), is effective when the selectivity of the foreign function is small.

![Figure 23. Q4 with $S_p = 0.3$ (10N, 10N/100N)](image)

Fig. 24 shows the cases of (100N, 10N) and (100N, 100N). The two-step processing method (tmp) is efficient for (100N, 100N).

![Figure 24. Q4 with $S_p = 0.3$ (100N, 10N/100N)](image)

6.3.4. Case of Q4 with $S_p = 0.1$
As in the case of Q3 with $S_p = 0.1$, the one-step processing method is better than the two-step processing method when we set the selectivity factor for the population attribute to be small. It is shown in Fig. 25 for (10N, 10N) and (10N, 100N). Similar behaviors can be observed for the cases of (100N, 10N) and (100N, 100N), but their differences tend to shrink.

6.3.5. Cases of Q1 and Q2
We briefly mention for the cases of Q1 and Q2. Among the three query processing methods, the two-step processing method (where) requires more processing time; the difference increases as the selectivity value of the foreign function increases. As we described before, this is because embedding key values into the condition part of SQL-2 is not efficient in such a case. With respect to the two-step processing method (tmp) and the one-step processing method, we have observed the tendency that the two-step processing method (tmp) has smaller cost when the selectivity of the foreign function is smaller. Except for the cases that the sizes of facility and location tables are small, the two-step processing method (tmp) is effective. Fig. 26 shows such an example. We can get similar graphs for other parameter settings.

![Figure 25. Q4 with $S_p = 0.1$: (10N, 10N/100N)](image)

![Figure 26. Q1 (100N, 10N/100N)](image)

6.4. Summary of Experiments
We can summarize the result in Subsection 6.3 as follows:

- The two-step processing method (where) is worse than the two-step processing method (tmp) in most situations: The reason is that PostgreSQL used in the experiments cannot handle many key values appearing in the SQL in operator.

- The processing cost of the one-step processing method does not depend to the selectivity of the foreign function: This is obvious from its processing method; once the required tuples are obtained from the RDBMS, the processing in the middleware. Therefore, the selectivity factor of the foreign function does not affect to the processing cost of SQL-3.
For queries that only include foreign function conditions (as in Q1 and Q2), the two-step processing method (tmp) is generally efficient. It is especially noticeable when the selectivity of the foreign function is small; if the selectivity is nearly equal to 1, the processing cost becomes to the similar level to the one-step processing method.

If the condition part includes other conditions in addition to the foreign function call (as in Q3 and Q4), the efficiency of the three approaches depends on the selectivity factors: If the selectivity of the foreign function is small and that of the other conditions are large, two-step processing method (tmp) is the most efficient method. In the opposite case, the one-step processing method is better.

If the processing cost of the RDBMS is quite small, the one-step processing method is efficient in general: Such cases occur, for example, when the target relational database is small or when the population conditions in Q3 and Q4 can filter out most of the unqualified tuples. Since the overall processing cost becomes small in these cases, the overhead of two interactions with the RDBMS required in the two-step processing method is not compensated in the total cost.

In summary, it can be concluded that the two-step processing method (tmp) is efficient for the queries such as Q1 and Q2. For Q3 and Q4, we need to select an appropriate one from the two-step processing method (tmp) and the one-step processing method considering the selectivities of the foreign function and other conditions. But we should note that the one-step processing method has advantage for the case that the processing cost in the RDBMS is small.

7. Conclusions and Future Work

In this paper, we have proposed query processing methods for XML view constructed over relational databases especially when queries include foreign function calls. We assumed the middleware and the underlying RDBMS cooperate to realize the XML view facility. Since foreign functions are evaluated on XML fragments materialized in the middleware, the previous approaches to XML view support is not applicable to our context. Therefore, we have extended the approach of the XPERANTO group to support XML view queries including user-defined foreign functions.

We have proposed two types of query processing methods, the two-step processing method and the one-step processing method. Moreover, we further proposed two approaches for the two-step processing method, the two-step processing method (where) and the two-step processing method (tmp). We have examined the performance of three methods using a conventional RDBMS and compared their effectiveness.

The future work includes the following issues:

- Broadening the supportable XML queries: We plan to reduce the constraint for the target XML queries shown in Subsection 4.1 to support broader class of queries. For this purpose, we need to generalize query processing method shown in Subsection 4.2.
- Query optimization: SQL queries shown in Figs. 16 through Fig. 19 are the result of the mechanical transformation processes and obviously contain redundant instructions. Although the underlying RDBMS can eliminate these redundancies by its optimization facility, it would be better to generate more simplified SQL queries.

Since XML query optimization in the middleware over RDBMSs are one of the hot research issues [4, 7], we may be able to include results of such approaches into our framework.

- Development of other query processing approaches: In the experiment, we used query processing facilities that are applicable to most of the off-the-shelf RDBMSs. However, we may need to consider to include advanced features, for example, the use of bitmap indexes for the efficient filtering and the use of WITH clause incorporated in SQL:1999 instead of view definitions.

- Selection of an appropriate query processing method: As shown in the experiment, the efficiency of the two-step processing method (tmp) and the one-step processing method depend on given queries and selectivity factors. We need to develop methods to select the optimal query processing method depending on the given situation to reduce query processing cost.

Also, we need to do our experiment on larger databases and other RDBMSs.

Acknowledgments

This research was supported in part by the Grant-in-Aid for Scientific Research from the Japan Society for the Promotion of Science and the Ministry of Education, Culture, Science, and Technology (#14019009 and #14780316).

References