Query Containment for XML-QL

Deepak Jindal, Sambavi Muthukrishnan, Omer Zaki
{jinjal, sambavi, ozaki}@cs.wisc.edu
University of Wisconsin-Madison
April 28, 2000

Abstract

The ability to answer an incoming query based on cached results of a previous query has been shown to lead to significant performance enhancements in the relational domain. We believe that similar gains can be made in the XML domain. To determine the feasibility of such gains we study query containment for XML-QL queries. We present a scheme and a Java implementation for query containment. We further integrate this scheme with the Niagara XML-QL query engine.

1. Introduction

Query containment is a central problem in many database and knowledge base applications especially query optimization. The problem of testing for query containment amounts to identifying a sufficient condition to recognize queries that are subsumed in an already executed query. We address this problem in the framework of the XML semi-structured data model. We propose a set of rules to identify if an XML-QL query is the superquery for a new XML-QL query. We assert that our query containment mechanism is conservative. Further, our query containment mechanism does not make use of the DTD (schema description) of the XML data in the process of containment testing. The DTD is built dynamically from the queries themselves, making our technique more flexible. This model ensures that query containment can be used in a broad spectrum of applications. For instance, it can be used at untrusted proxies, which do not have access to the DTD from the server.

2. Methodology

We approach query containment for XML-QL by first identifying an internal representation for the incoming query. We call this internal representation the Normal Form. Mechanisms were then investigated to convert the incoming query efficiently into our internal form. Once we had this internal representation we came up with rules to check whether a new query was contained in a previously cached query. For efficient retrieval of cached queries we used a simple indexing scheme. Finally, to evaluate the feasibility of query containment we implemented our scheme in Java and integrated it with the Niagara XML query engine.
3. XML-QL Grammar

In our work we use a subset of XML-QL defined in [1]. Given below is the grammar for the subset we studied.

\[
\begin{align*}
\text{XML-QL} & : = \text{QueryBlock} \ <\text{EOF}> \\
\text{Query} & : = \text{Element} | <\text{STRING}> | <\text{VAR}> | \text{QueryBlock} \\
\text{Element} & : = \text{StartTag} \ \text{Query} \ \text{EndTag} \\
\text{StartTag} & : = '<<\text{ID}>\text{SkolemID}\text{Attribute}*'>' \\
\text{SkolemID} & : = '<\text{ID}> '(' <\text{VAR}> (', <\text{VAR}>)* ')'>' \\
\text{Attribute} & : = '<\text{ID}> '=' ('" '<\text{STRING}> '" | <\text{VAR}> )' \\
\text{EndTag} & : = '<'/ <\text{ID}>? '>' \\
\text{QueryBlock} & : = \text{Where} \ \text{Construct} \\
\text{Where} & : = 'WHERE' \ \text{Condition} (', \ \text{Condition})* \\
\text{Construct} & : = 'CONSTRUCT' \ \text{Query} \\
\text{Condition} & : = \text{Pattern BindingAs? 'IN' DataSource} \\
\text{Pattern} & : = \text{StartTagPattern Pattern* EndTag} | <\text{ID}> | <\text{VAR}> \\
\text{StartTagPattern} & : = '<' <\text{ID}> \ \text{Attribute}* '>' \\
\text{BindingAs} & : = 'CONTENT\_AS' <\text{VAR}> \\
\text{DataSource} & : = <\text{VAR}> | <\text{URI}> \\
\end{align*}
\]

We chose this subset to make our query containment simple. We focussed on basic features of XML-QL like multiple WHERE clauses, joins, nested queries, skolem functions and CONTENT\_AS bindings.

XML-QL allows two different ways to do GROUP\_BY operation of SQL

- Nested queries
- Skolem functions

Both of them are equivalent. Each can be converted into the other without changing the meaning of the query. The following example shows this equivalence.

Example 1

\[
\begin{align*}
\text{WHERE} <\text{bib}> \\
<\text{author}> $a </> \\
<\text{age}> $y <</> \\
<</> \ \text{CONTENT\_AS} \ $p \ \text{IN "bib.xml"} \\
\text{CONSTRUCT} <\text{result}> \\
<\text{author}> $a </> \\
<\text{age}> $y <</> \\
<</> \ \text{WHERE} <\text{book}> $b </> \ \text{IN} \ $p \\
\text{CONSTRUCT} <\text{document}> $b <</> \\
<</>
\end{align*}
\]

is equivalent to
Both will produce results which are grouped by author and age. We use this property to convert nested queries to Normal Form (described next).

4. Normal Form

All queries are stored internally in Normal Form. Normal Form requires that we divide every query into a WHERE part and CONSTRUCT part and store the two as separate trees. The aim of the Normal form is to give a standard representation to any query and to ease the process of containment checking. A query in Normal form must satisfy the following conditions:

1. There are no nested queries.
2. Data source of a condition in the where part must be a Uniform Resource identifier (URI) having a form similar to "www.a.b.c/bib.xml".

In addition, our implementation of the Normal form has the following features with respect to the where part of the query which further aid the query containment phase.

- Associated with every non-leaf node is a boolean variable hasStar that specifies whether all possible descendants of this node occur in the result or not.
- Associated with every node, be it leaf or non-leaf, is a boolean variable inResult that specifies whether the node or any of its children is present in the result or not.
- All children of a node are sorted alphabetically on their start tag value. All attributes of a node are also sorted alphabetically.

Example 2

A query has two parts. One pattern matching tree (for the Where clause) and the other output format tree (for the Construct clause)

WHERE

<book>
  <publisher><name>Addison-Wesley</name></publisher>
  <title>$t</title>
  <author>$a</author>
</book> IN "www.a.b.c/bib.xml"

CONSTRUCT

<Result>
  <author> $a</author>
  <title>$t</title>
</Result>
Example 3

Consider the following query having a nested sub-query.

```
WHERE
   <book> $p </> IN www.a.b.c/bib.xml,
   <title>$t</> IN $p,
   <publisher><name>Addison-Wesley</></> IN $p
CONSTRUCT
   <result>
      <title> $t</>
      WHERE
         <author> $a </> IN $p
      CONSTRUCT
      <author> $a</>
   </>

(or equivalent query)

WHERE
   <book> $t</>
   <title>$t</>
   <publisher><name>Addison-Wesley</></>
   </> CONTENT_AS $p IN "www.a.b.c/bib.xml"
CONSTRUCT
   <result>
      <title> $t</>
      WHERE
         <author> $a </> IN $p
      CONSTRUCT
      <author> $a</>
   </>
```
Normal form representation:

```
WHERE
“a.b.c/bib.xml”

<book>
  <title> $t </>
  <author> $a </>
  <publisher><name>Addison-Wesley</></>
</>
```

CONSTRUCT
```
<result; ID = titleID($t)>
  <title> $t </>
  <author> $a </>
</>
```

Addison-Wesley

XML-QL query representation derived from Normal form tree given above after unnesting:

```
WHERE

<book>
  <title> $t </>
  <author> $a </>
  <publisher><name>Addison-Wesley</></>
</>
```

```
IN “www.a.b.c/bib.xml”
```

```
CONSTRUCT

<result; ID = titleID($t)>
  <title> $t </>
  <author> $a </>
</>
```

With Normal Form, it becomes easy to check for containment. The Normal form trees for examples 2 and 3 are isomorphic to each other. So one query can be answered from the other.

Example 4

Now we look at a join query. A Join query has a number of simple queries like the ones in example 2 and 3 that are joined on some elements.

```
WHERE

<article>
  <author>
    <firstname> $f </> // firstname $f
    <lastname> $l </> // lastname $l
  </>
</>

CONTENT AS $a IN “www.a.b.c/bib.xml”

<book year = $y>
  <author>
    <firstname> $f </> // firstname $f
    <lastname> $l </> // lastname $l
  </>
</>
```

```
IN “www.a.b.c/bib.xml”
```

```
y > 1995
```

```
CONSTRUCT

<article> $a </>
```

```
(or equivalent query)

WHERE

    <article>
    <author>
        <firstname> $f </> // firstname $f
        <lastname> $l </> // lastname $l
    </author>
    </article>

WHERE-1 WHERE-2 CONSTRUCT

    "a.b.c/bib.xml" "a.b.c/bib.xml"

                  *                  *

WHERE-1 WHERE-2 CONSTRUCT

    <article>
    <book year = $y>
    <author>
        <firstname> $f </> // firstname $f
        <lastname> $l </> // lastname $l
    </author>
    </book;

    "www.a.b.c/bib.xml"

WHERE-1 WHERE-2 CONSTRUCT

          *          *

\ y > 1995

CONSTRUCT $e

In this case the representation will contain two trees for the where part. Join is implicitly represented by matching variable names.

Normal form representation:

```
WHERE-1 WHERE-2 CONSTRUCT

"a.b.c/bib.xml" "a.b.c/bib.xml"

'article'          'book; [y > 1995]'                'article'

'author'           'author'                              'author'

'firstname'        'firstname'                           'firstname'

'lastname'         'lastname'                           'lastname'

$f   $l          $f   $l          $f   $l

*' is used to represent the rest of the children nodes available due to the CONTENT_AS binding. However hasStar is set to true only when the binding variable is generated in the CONSTRUCT part of the query.

**Converting to Normal form**

While converting to Normal form, one needs to ensure that the meaning of the query does not change. The following steps transform a given query into the Normal form.

1. Unnesting of queries

2. Variable data source resolution

3. Set hasStar for the query tree nodes
4. Set inResult for the query tree nodes

5. Miscellaneous sorting

**Unnesting of queries**

Nested queries are those that have their WHERE clause inside the CONSTRUCT clause of a parent query. These queries are used to achieve the same result as GROUP BY in SQL. Since queries can be nested to any level, query containment becomes difficult with nested queries (or more conservative if we try to keep it simple). Therefore we convert a nested query into an equivalent query which uses skolem functions. This is done in two steps: Skolemization and Unnesting.

**Skolemization**

Skolemization adds redundant skolem functions to all the nodes in the CONSTRUCT part of the query. It puts skolem functions in all nodes such that the meaning of the query remains the same. It starts by assigning skolem functions to leaf nodes with a single variable. Going bottom up, it then propagates skolem functions of all the children nodes to the parent node. If the child involves a nested query, no skolem function is propagated. The function Annotate given below outlines our procedure for Skolemization.

Annotate(Construct construct){
    //Go bottom up the construct tree and do the following:
    If the current node is a leaf:
        if node is a variable: add that variable in the skolem function of node.
        else if the node is a constant: do nothing.
        else if the node is a nested query: Annotate its construct part recursively.
    else:
        for each of node’s children Annotate their construct part recursively.
        add the variables in node’s children’s skolem function to node’s skolem function.
}

The following example gives the result of running Annotate on a query:

**Example 5**

**Input query**

```xml
WHERE <bib>
    <author> $a </>
    <age> $y </>
</> CONTENT_AS $p IN "bib.xml"
CONSTRUCT
    <result>
        <author> $a </>
        <age> $y </>
        WHERE <book> $b </> IN $p
        CONSTRUCT <document> $b </>
```
Output Query

WHERE <bib>
    <author> $a ///</author>
    <age> $y ///</age>
</bib>
CONTENT_AS $p IN "bib.xml"

CONSTRUCT

WHERE
</bib>
CONTENT_AS $p IN "bib.xml"

CONSTRUCT
    <result; ID = resultID($a,$y)> $a ///</result>
    <age; ID = ageID($y)> $y ///</age>
WHERE
    <book> $b ///</book> IN $p

CONSTRUCT
    <document; ID = documentID($b)> $b ///</document>

Unnesting

After skolemization, unnesting simply removes nested WHERE clauses and appends them in the primary WHERE clause. The primary WHERE clause is the only WHERE clause which is not part of the CONSTRUCT clause. The nested query is replaced by its own CONSTRUCT part. It uses the following two functions, which call each other recursively.

Query unNest(Query query){

    // get the construct clause from the query.
    Construct construct = query.extractConstructClause();

    // Remove all nested where conditions.
    Vector newConditions =
        construct.getNestedConditions(newConditions);

    // Add these conditions to the where part of original query.
    query.add(newConditions);

    return query;
}

// This function removes nested WHERE conditions from the construct
// clause and replaces them with the corresponding construct clauses.
Vector getNestedConditions(Vector initialConditions){

    For each nested WHERE condition (=where) in the current construct clause{

        // recursively un-nest the where condition.
        unnestedWherePart = unNest(where);

        // add conditions to the input vector.
        initialConditions.add(unnestedWherePart.getConditions());

        // replace the nested WHERE clause in the current construct
        // clause by construct clause of the unnestedWherePart.
        ReplaceWhereByConstruct
            (unnestedWherePart.getConstructClause());
    }
return initialConditions;
}

The following example illustrate how unnesting works.

Example 6

Input query
Same as the output query from Example 5.

Output query

WHERE <bib>
  <author> $a </>
  <age> $y </>
  </> CONTENT_AS $p IN "bib.xml",
  <book> $b </> IN $p
CONSTRUCT
  <result; resultID($a,$y)>
    <author; authorID($a)> $a </>
    <age; ageID($y)> $y </>
    <document; documentID($b)> $b </>
  </>

Variable data source resolution

Typically a data source can either be

1. a URI or Uniform Resource identifier having the form "www.a.b.c/bib.xml"
or
2. a variable that is defined in another condition of the where part. The variable is either the binding
data of a node or the variable in a leaf node in the other condition.

In our Normal form, there should not be any condition having a data source that is a variable. This is
because each such variable can be "resolved" and merged into the condition defining that variable. This
process is known as "variable data source resolution" which we describe in this section. Consider the
following naïve query that does not do much but, makes for a simple example.

Example 7

WHERE
  <book>
    <publisher>$p</>
  </> in "www.a.b.c/bib.xml",
  <name> Addison-Wesley</> in $p
CONSTRUCT $p

The where part of this query has 2 conditions.
1.  \[
\begin{align*}
\text{<book>}
\text{  <publisher>$p$/>}
\text{  }$
\text{</} > \text{ in "www.a.b.c/bib.xml"}
\end{align*}
\]

2.  \[
\begin{align*}
\text{<name> Addison-Wesley</> in $p}
\end{align*}
\]

The data source of condition 2 is $p$ which is resolved and found in condition 1. Thus, condition 2 must be merged into condition 1. Upon merging we get one condition:

\[
\begin{align*}
\text{<book>}
\text{  <publisher>}
\text{    <name>Addison-Wesley</>}
\text{  }$
\text{</} > \text{ in "www.a.b.c/bib.xml"}
\end{align*}
\]

Such a compact representation reduces complexity of query containment making it tractable. So finally after resolution of variable data source the original query given in example 7 results in

WHERE

\[
\begin{align*}
\text{<book>}
\text{  <publisher>}
\text{    <name>Addison-Wesley</>}
\text{  }$
\text{</} > \text{ in "www.a.b.c/bib.xml"}
\end{align*}
\]

CONSTRUCT $p$

The reader may have observed that in the resolution process the leaf node \(<\text{publisher}>$p</> transforms into a non-leaf or "internal" node and $p$ disappears. However, $p$ is present in the result making the query given above illegal. In order to remedy this, whenever a leaf node is transformed into an internal node, we cause the variable of the leaf node to become the "CONTENT_AS" binding data variable of the internal node. In doing so, if there already exists a binding data variable in the leaf node, all instances of that variable are replaced with the newly promoted variable. This copying of a variable into the binding data does not modify the meaning or results of the query. Using the “CONTENT_AS” remedy, the resolution process will result in the following query.

WHERE

\[
\begin{align*}
\text{<book>}
\text{  <publisher>}
\text{    <name>Addison-Wesley</>}
\text{  }$
\text{  CONTENT_AS $p$}
\text{  }$
\text{</} > \text{ in "www.a.b.c/bib.xml"}
\end{align*}
\]

CONSTRUCT $p$

The algorithm given below sums up this process. To perform variable data source resolution, one only needs to look at the conditions in the where part of the query.

Let \(Q = \) given query written in XML-QL.

1. Obtain the list of conditions in the where part of \(Q\). Call this \(\text{list}_{\text{cond}}\).

2. Reorganize elements of \(\text{list}_{\text{cond}}\) into two lists \(\text{list}_{\text{URI}}\) and \(\text{list}_{\text{var}}\) such that \(\text{list}_{\text{URI}}\) contains conditions whose data sources are URIs and \(\text{list}_{\text{var}}\) contains conditions whose data sources are variables.

3. For each element condition \(\text{cond}_{\text{var}}\) of \(\text{list}_{\text{var}}\)
a) Resolve its data source d by finding a condition cond_{URI} of list_{URI} which defines d.

b) Merge cond_{URI} into cond_{URI}.

4. Make list_{URI} the new list of conditions for the where part of Q.

Set hasStar

Upon completion of variable data source resolution phase we are left with a query with a where part that contains one or more conditions each having a single URI datasource. Note that a single URI can be the data source for more than one condition. This is a common occurrence especially when one wants to perform joins. As mentioned before, every condition can be represented by a tree. Associated with every non-leaf node of a condition tree is a boolean variable hasStar. If hasStar is set to true it implies that this node is bound to a CONTENT_AS variable and hence contains all possible children. This information is necessary when comparing two queries and checking whether one is contained in the other. The following example will help make this clear.

Query 1

WHERE

<book>
  <publisher>
    <name>Addison-Wesley</name>
  </publisher>
  content_as $p
</book> in "www.a.b.c/bib.xml"
CONSTRUCT $p

Query 2

WHERE

<book>
  <publisher>
    <name>Addison-Wesley</name>
    <city>Madison</city>
  </publisher>
  content_as $p
</book> in "www.a.b.c/bib.xml"
CONSTRUCT $p

Consider the tree form of the two queries given above after conversion to Normal form.

If Query2 was isomorphic to Query1 then we could have said that Query2 is contained in Query1. However, the two queries given above are not isomorphic since Query2 has an additional child with tag <city> so the naive conclusion is that Query2 is not contained in Query1.
The `<publisher>` node of Query1 is bound to CONTENT AS $p$ which is generated as part of the result (construct $p$). This implies that in actuality the `<publisher>` node contains all its children including `<city>`. This is indicated by setting the corresponding hasStar to true (indicated by a * in the tree for Query1). With the hasStar information we conclude that Query2 is contained in Query1 which is the case. This demonstrates the use of hasStar.

To accurately make a decision whether hasStar must be set to true or not one has to check whether the associated binding variable is part of the result or not. In the above case before setting hasStar of `<publisher>` to true we check to see if $p$ is part of result. It is only after this that we set hasStar to true.

To decide what value to set hasStar to for a node $n$ we use the following algorithm (by default before this setting is done, all nodes have hasStar set to false).

1. Is $n$ an internal-node? If not quit
2. Obtain binding data variable $v$ for $n$
3. If $v$ is contained in result then
   1. set hasStar to true.
   2. recursively descend all sub-trees rooted at $n$ setting hasStar of all non-leaf nodes to true.

Set inResult

In the Normal form a variable called inResult is associated with every node (leaf or non-leaf) of a condition tree. A true value signifies that some variable in the tree rooted at this node is present in the result. This information helps to determine whether the given query is not contained in another query. The following queries show this.

Consider the following two queries.

Query 1

```
WHERE
  <book>
    <publisher>
      <city>$c</>
      <address>$a</>
    </publisher>
  </book>
</>
in "www.a.b.c/bib.xml"
CONSTRUCT
  <result>
    <city>$c</>
  </result>
</>
```
Query 2

WHERE
<book>
  <publisher>
    <city>$c</city>
    <address>$a</address>
  </publisher>
</book>
in "www.a.b.c/bib.xml"

CONSTRUCT
<result>
  <city>$c</city>
  <address>$a</address>
</result>

Their tree representation of where parts is given below.

![Tree Representation](image)

Even though the where trees of both queries are isomorphic, Query2 is not contained in Query1. This is so because Query1 only has the <city> variable as part of its result and not the <address> variable. Thus there is insufficient information to answer Query2 which also has the <address> variable as part of its result. When examining the <address> node of Query2 during query-containment we would see that the <address> node has inResult = false in Query1. This would be sufficient for us to conclude that Query2 is not contained in Query1.

**Miscellaneous Sorting**

In addition to variable data source resolution, setting of hasStar and inResult, the Normal form also demands that the following be true for all condition trees.

1. All children of an internal node are sorted alphabetically on the value of the tag variable of the children nodes.

2. All attributes of a node are sorted alphabetically on the attribute name.
5. Query Containment Checking

Query containment addresses the problem of checking if the tuples that result on executing one query is always a subset of the tuples generated on executing another (base) query.

**Query containment rules:**

**Terminology**

*Input* is the new query to be checked for containment in the old query *base*. Every component of *base* is represented as *base.<component>*. We use the following representation for the different components.

- *where*: It is used to represent the WHERE clause.
- *construct*: It is used to represent the CONSTRUCT clause.
- *node*: It is used to represent any node (leaf or internal) in *where* or *construct*.
- *variable*: It is used to represent a leaf node which is a variable.
- *bindingVariable*: It is used to represent a variable that is bound to an internal node using CONTENT_AS.
- *data*: It is used to represent the data (variable name or identifier) in a leaf node.
- *dataSources*: It is used to represent the data sources used in a condition.
- *condition*: It is used to represent one IN clause in *where*.
- *attributeList*: It is used to represent the list of attributes in a node.

**Assumptions**

1. *Input* and *Base* are represented in Normal Form.
2. A condition cannot have multiple children with the same tag identifier. For example, the following query is illegal.

**Example 8**

```xml
WHERE
  <book>
    <title> Readings in Database Systems </title>
    <author> $author1 </author>
    <author> $author2 </author>
  </>
  IN "www.a.b.c/bib.xml"

CONSTRUCT
  <result>
    $author1
  </>
```

**Data structures used**

1. *Symbol Table*: When we pattern match *input.where* with *base.where* we need to store the mapping of variables in *base* to nodes in *input*. This table does exactly this. It contains two fields for each mapping
   - *base.variable*
   - *input.node*

   This is built up during the query containment checking process. If query containment succeeds, this table is used for the generation of the new XML-QL query to be run on the XML result file produced by *base*. 

---

14
2. *Reverse Symbol Table:* This table contains a list of all variable names, which appear in the *where* part of the *input* query. This table is built during query containment checking. It contains exactly one entry per *input.variable*. Its purpose is to handle joins which is explained later.

*Functions defined on these tables*

a) AddSymbolTable(*base.variable, input.node*)

This method adds a record with the above entries to the Symbol Table. It fails if there was already a mapping, which conflicts with the new mapping. New mapping conflicts with the old mapping in two cases

- *input.node* is not a leaf in the old mapping or the new mapping.
- *input.data* is different in the old mapping and the new mapping.

It also adds *input.variable* to Reverse Symbol Table if *input.node* is a leaf and the mapping is new. If this method fails, query containment automatically fails.

b) AddReverseTable(*input.node*)

This method adds all *input.variables* in the subtree corresponding to *input.node* (including *bindingVariables*) to the list of variables in the Reverse Symbol Table. It fails if there is already an entry for any *input.variable* in the table. If this method fails, query containment automatically fails.

*Example 9*

The following example shows how the above tables are built up. In this example, *input* query is contained in *base* query.

**Base query:**

```xml
WHERE
<brick>
  <title> $title </>
  <author> $author </>
  CONTENT_AS $p
  <year> $year </>
</> IN ”www.a.b.c/bib.xml”
CONSTRUCT
<result>
  <author> $author </>
  <title> $title </>
</>
```

**Input query:**

```xml
WHERE
  <book>
    <title> Ulysses </>
    <author>
      <firstname>$first </>
      <lastname> $last </>
    </author>
    CONTENT_AS $p1
    <year> $newyear </>
  </>
  IN “www.a.b.c/bib.xml”
CONSTRUCT
<result>
  <author> $first </>
  <year> $newyear </>
</>
```

---

<table>
<thead>
<tr>
<th>input.Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$first</td>
</tr>
<tr>
<td>$last</td>
</tr>
<tr>
<td>$newyear</td>
</tr>
<tr>
<td>$p1</td>
</tr>
</tbody>
</table>
Symbol Table

<table>
<thead>
<tr>
<th>$title</th>
<th>&lt;title&gt; Ulysses &lt;/&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$author</td>
<td>&lt;author&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;firstname&gt; $first &lt;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;lastname&gt; $last &lt;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/&gt; CONTENT_AS $p1</td>
</tr>
<tr>
<td>$year</td>
<td>&lt;year&gt; $newyear &lt;/&gt;</td>
</tr>
<tr>
<td>$p</td>
<td>&lt;author&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;firstname&gt; $first &lt;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;lastname&gt; $last &lt;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/&gt; CONTENT_AS $p1</td>
</tr>
</tbody>
</table>

Query Containment Rules

A detailed explanation of the rules follows in the next section.

Rules for Query containment

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of $base.conditions = Number of $input.conditions</td>
<td>Nil</td>
</tr>
<tr>
<td>Every $input.condition is contained in a corresponding $base.condition. The relationship between the two is a one to one mapping.</td>
<td>Nil</td>
</tr>
</tbody>
</table>

Rules for Condition containment

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$base.dataSources = $input.dataSources</td>
<td>Nil</td>
</tr>
<tr>
<td>$input.rootnode is contained in $base.rootnode</td>
<td>Nil</td>
</tr>
</tbody>
</table>
Rules for Node containment

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>If base.node has a bindingVariable</td>
<td>AddSymbolTable(base.bindingVariable, input.node)</td>
</tr>
<tr>
<td>input.attributelist is contained in base.attributeList</td>
<td>Nil</td>
</tr>
<tr>
<td>input.node has inResult set to true =&gt; base.node has inResult set to true</td>
<td>Nil</td>
</tr>
<tr>
<td>If base.node is a leaf node containing a variable and</td>
<td></td>
</tr>
<tr>
<td>a) input.node is a leaf node containing a variable (or)</td>
<td>AddSymbolTable(base.variable, input.node)</td>
</tr>
<tr>
<td>b) input.node is a leaf node containing an identifier =&gt; base.node should have inResult = true (or)</td>
<td>AddSymbolTable(base.variable, input.node)</td>
</tr>
<tr>
<td>c) input.node is an internal node containing a subtree =&gt; base.node should have inResult = true</td>
<td>AddReverseTable(input.variable)</td>
</tr>
<tr>
<td>If base.node is a leaf node containing an identifier =&gt; input.node is a leaf node containing the same identifier</td>
<td></td>
</tr>
<tr>
<td>If base.node is an internal node =&gt;</td>
<td></td>
</tr>
<tr>
<td>a) input.node is an internal node (and)</td>
<td>Nil</td>
</tr>
<tr>
<td>b) input.node has equal or more children than base.node (and)</td>
<td>Nil</td>
</tr>
<tr>
<td>c) If input.node has hasStar set to true =&gt; base.node has hasStar set to true (and)</td>
<td>Nil</td>
</tr>
<tr>
<td>d) Either every child node of input.node is “node contained” in some child node of base.node (or)</td>
<td>AddReverseTable(input.node’s child node) for all child nodes of input.node which are not contained in some child node of base.node</td>
</tr>
<tr>
<td>base.node has hasStar set to true</td>
<td></td>
</tr>
<tr>
<td>e) Every child node of base.node has exactly one input.node child node “node contained” in it</td>
<td></td>
</tr>
</tbody>
</table>

Explanation of the rules

Top Level Rules

1. Number of base.conditions = Number of input.conditions.

   If input has fewer conditions, it becomes less restrictive than base. It may involve result XML data that satisfy fewer conditions than data that satisfy conditions in base. If input has more conditions, it may
involve more Data Sources than base. It may include XML data that base did not even consider. Hence containment fails in either of the above two cases

2. There is a one-one correspondence between base.conditions and their corresponding “condition contained” input.conditions.

This is to ensure that every single base.condition is satisfied by input too. If this were not true, a single base.condition could “condition contain” multiple input.conditions leading to the same case as in Rule 1 (input being less restrictive than base).

Rules for Condition containment

1. base.dataSources = input.dataSources.
   This rule ensures that both input and base are using the same data.

2. input.rootnode is contained in base.rootnode.
   This is for containment for the particular root nodes in this condition.

Rules for Node containment

1. If base.node has a bindingVariable, make an entry (base.bindingVariable, input.node) in the Symbol Table.

   This rule and its corresponding action ensure proper handling of CONTENT_AS binding variables. By binding the base CONTENT_AS variable to the corresponding CONTENT_AS variable (if present) in input and the corresponding subtree in input, variable bindings are properly maintained for generating the result query in case query containment succeeds.

2. input.attributeList is “attribute contained” in base.attributeList. In other words,
   If base.attributeList = A₁, A₂, …, Aₖ (and)
   input.attributeList = B₁, B₂, …, Bₖ
   then every attribute Aᵢ, 1 <= i <= k, should have an exactly equal attribute in Bᵢ, 1 <= i <= k where Aᵢ and Bᵢ equality is implied when the attribute names are identical and the values are also equal.

3. input.node has inResult set to true => base.node has inResult set to true

   This rule ensures that any variable in input, which is generated in the CONSTRUCT part is also present in the result generated by base.

4. If base.node is a leaf node and contains a variable:
   a) If input.node is a variable, there is a direct match. The base.node variable need not be present in the base query result XML data unless the input.node variable is present in input result data.
   b) In case, input.node contains an identifier, to make sure this mapping is not lost while generating the new query to run on base result data, it is essential that base.node has inResult set to true.
   c) If input.node forms the root of a subtree, the same case as in (b) above holds. The mapping should not be lost and so base.node should have inResult set to true. Further in this case, all input variables occurring in the subtree rooted at input.node should be entered into the Reverse Symbol Table.

   In all the above cases, since base.node corresponds to a variable, the variable binding has to be entered in the SymbolTable.

6. If base.node is a leaf node and contains an identifier, input.node should also be a leaf node containing the same identifier.
7. If base.node is an internal node, the following conditions should hold:
   
   i. input.node is not a leaf node
      
      If input.node were allowed to be a leaf node, then input becomes less restrictive than base and hence query containment cannot succeed.
   
   ii. input.node has equal or more children than base.node.
      
      If input.node has fewer children than base.node, again input becomes less restrictive than base.
   
   iii. If input.node has hasStar set to true => base.node has hasStar set to true.
      
      If input.node has hasStar set to true, it implies that input.node is bound to a CONTENT_AS variable, which appears in the result. To ensure that this variable can be generated from the result XML data of base, base.node should also be bound to a CONTENT_AS variable that appears in the base result data.
   
   iv. Either every child node of input.node is “node contained” in some child node of base.node (or) base.node has hasStar set to true.
      
      If a child node of input.node cannot be directly “node contained” in a child node of base.node, the input.node can be allowed to be contained if base.node is bound to a CONTENT_AS variable which appears in the result of base. In this case, all variables appearing in the subtree rooted at the child of input.node in question should be entered in the Reverse Symbol Table.
   
   v. Every child node of base.node has exactly one input.node child node “node contained” in it.
      
      Given that child tag identifiers are unique (by our assumption), this ensures that every base.node child has a matching/contained input child node so that input is not less restrictive than base.

6. Generation of the new XML-QL query

Once query containment is determined, it is necessary to generate a new XML-QL query to run on the XML data generated as a result of execution of the base query. Assume the same terminology as used in the Query Containment rules section. This query is generated as follows:

1. The WHERE part of the new query is generated from the CONSTRUCT part of the base query. The base.construct tree is traversed, replacing all variables using the bindings maintained in the Symbol Table. A variable occurrence is replaced as follows:
   
   If a lookup in the Symbol Table yields both a new variable and a subtree, the subtree is used to replace the variable in the WHERE tree and the new variable is included as a CONTENT_AS binding variable for the node in question.
   
   If it yields only a new variable, the new variable simply replaces the original one in the query tree.
   
   Alternatively, if the lookup yields only a subtree, the subtree replaces the original variable.

2. The Construct part of the new query is simply the input.construct tree (which is in Normal form).

3. The data source for the query is the result XML file for base.

This query as generated above can now be passed on to a query engine.
7. Handling of Joins

There are two issues in the handling of joins. One is to make sure that if base has a join, then a corresponding join in input, which is meaningful should be allowed. At the same time, any new joins in input should lead to query containment failure.

The first case is handled in the way entries are made in the Symbol Table. Whenever a new entry is added to the Symbol Table, a check is made to see if there is another entry for the same base.variable. If there is, the new mapping must be checked to see if it leads to a meaningful join. The only meaningful and containable joins are when the input.nodes to which base.variable maps in the two mappings are leaf nodes containing identical data (variables or identifiers). Even if base.variable maps to two identical subtrees in input.node, the join is not ensured to be meaningful because the subtree may look only at a subset of all possible child nodes of input.node.

The second case of new joins having to be disallowed is handled by using the Reverse Symbol Table. By ensuring that all variables in input (except when the Symbol Table identifies a join) are entered in the Reverse Symbol Table, any invalid reoccurrence of a variable in input is identified during addition of entries to the Reverse Symbol Table. Consider the following example.

Example 10

Base query:

```xml
WHERE
<book>
  <title> $title </>
  <author> $author </>
  <coauthor> $coauthor </>
</> IN "www.a.b.c/bib.xml"
CONSTRUCT
<result>
  <author> $author </>
  <title> $title </>
</>
```

Input query:

```xml
WHERE
<book>
  <title>Ulysses </>
  <author>
    <firstname>$first</>
    <lastname>$last </>
  </>
  <coauthor>$first</>
</> IN "www.a.b.c/bib.xml"
CONSTRUCT
<result>
  <author> $first </>
</>
```

Here input is not contained in base for the reason that a new join is introduced which cannot be handled using the base query (since base cannot be ensured to contain $coauthor in its CONSTRUCT part and hence the join information may be lost in generating a new query).

8. Some interesting query containment cases

In this section we present some interesting query containment cases which are recognized by our rules. The rules can be worked through to verify these cases after the queries are converted to our Normal form. In both examples the input query is contained in the corresponding base query.
Example 11

**Base query**

```xml
WHERE
<bib>
  <vendor>$v</>
</bib>
IN "www.xmldata.com/data.xml"
CONSTRUCT
<result>
  <vendor>$v</>
</result>
```

**Input query**

```xml
WHERE
<bib> $p </bib>
  <vendor>$v1</> IN $p,
  <vendor><name>$n</></> IN $p
CONSTRUCT
<result>
  <name> $n </>
  WHERE <vendor><email>$e</></>
</result>
```

Example 12

**Base query**

```xml
WHERE
<bib>
  <vendor>
    <name>Amazon</>
    <book>
      <author>
        <firstname> $f</>
      </author>
    </book>
  </vendor>
</bib>
IN "www.xmldata.com/data.xml"
CONSTRUCT
<result>
  <book> $b </>
  WHERE <vendor><email>$e</></>
  IN $p
  CONSTRUCT <email> $e</>
</result>
```

**Input query**

```xml
WHERE
<bib>
  <vendor>
    <name>Amazon</>
  </vendor>
</bib>
  <book>
    <title> $t </>
    <author>
      <firstname> $f</>
      <lastname> Stevens</>
    </author>
  </book>
  WHERE <vendor><email>$e</></>
  IN $p
  CONSTRUCT <email> $e</>
</result>
```
9. Integration with Niagara

Niagara is an XML-QL query engine being developed at the University of Wisconsin-Madison. To demonstrate the correctness of our scheme we integrated our query containment mechanism with the existing Niagara query engine. Niagara has a client-server architecture. We integrated our code at the server side. Old queries with their results are cached at the server. Whenever a new query arrives, old queries are searched for possible containment. If a new query is contained in an old query, Niagara executes a new query on the cached file instead of fetching data for the URI from a possible remote machine.

To minimize search time and for efficient query containment checking we use indexing. Currently we index on data sources used in the query. For cache replacement we use a simple LRU scheme.

At present, Niagara does not support nested queries. However, we extended its parser to support nested queries. Since all queries are first converted into Normal Form, Niagara engine does not deal with nested queries. Also, Niagara does not support skolem functions. As a consequence we are not able to demonstrate nested queries on Niagara.

10. Conclusion

We present a scheme for query containment in XML-QL. Though we use a subset of XML-QL, most of the important features of the language were adequately considered. We believe that extending our scheme to include other features will not be too hard. Implementation of query containment with an existing query engine further demonstrates the promise of our scheme.

11. Future Work

Future work in this area can be in a myriad of directions. One can try to enhance our scheme to include the remaining features of XML-QL. We do not provide a formal proof for the correctness of our scheme. Proving it formally may be a challenging task. To make a convincing case for query containment having real world work loads would greatly help. This may not be practical given the infancy of XML. Another alternative would be to come up with artificial work loads resembling existing relational database query loads. These work loads may help determine how much overhead we can pay for query containment without greatly reducing the overall throughput of the system. As of now we try to answer a new query based on the results of one other cached query that contains it. It will be worthwhile to come up with a scheme to answer a new query based on the results of several other cached queries where none of these queries actually contain the new query in its entirety but rather contain only a part of it.

Bibliography