MORE ON THE INFORMATION CONTENT OF AN XML DATABASE

by
John Matthew Bannatyne Cocking

A Thesis Submitted to
the Faculty of the Graduate School at
The University of North Carolina at Greensboro
in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Greensboro
2004

Approved by

Committee Chair

To Stagger Lee and the Seven Missing Hearts

APPROVAL PAGE

This thesis has been approved by the following committee of the
Faculty of The Graduate School at The University of North Carolina at Greensboro

Committee Chair ______________________________

Committee Members ______________________________

____

____
Date of Acceptance by Committee

____
Date of Final Oral Examination

ACKNOWLEDGEMENTS

Thanks to Dr Sadri, Dr Fu and Dr Green. Partial funding was provided by NSF grant number IIS-0083312.

TABLE OF CONTENTS

Page

CHAPTER

I. INTRODUCTION……………………………………………………………..1

II. PREVIOUS WORK…………………………………………………………...4

III. CONTRIBUTIONS…….……………………………………………………..9

Improvements to Algorithms: Algorithm 4.1 (Translation)………………9
Data Structure..………………………………………………..…9
Ambiguity……..………………………………………………..10
SQL AS Keyword………..……………………………………..13
Aggregate Queries………………………………………………………16
Unified Algorithm for Translating Queries on HICT…………....17
Example 1……………………………………………………..18
Example 2……………………………………………………..25
Example 3……………………………………………………..31

IV. RELATED WORK…………………………………………………………..37

Schema-Free XQuery…………………………………………………37
XSearch: A Semantic Search Engine for XML……………………….....37

V. CONCLUSIONS AND FUTURE WORK…………………………………...38

Nulls and Missing Data - Two- vs. Three-Way Logic…………………...38
Multi-document Queries………………………………………………39

REFERENCES…………………………………………………………………..41

APPENDIX A: SUPPLIERSALT.XML………………..…………………………42

APPENDIX B: SUPPLIERSALT2.XML..…………………………………………44

APPENDIX C: SUPPLIERSALT3.XML…..………………………………………46

CHAPTER I

INTRODUCTION

XML is a text-markup method originally designed for large-scale electronic publishing ventures, but has since become an increasingly popular format for data exchange[1]. Overseen by the World Wide Web Consortium (W3C), a group concerned with furthering the reach and depth of the World Wide Web[2], XML has grown to the point where most major relational database systems offer some level of XML integration, usually by allowing the users to deal with their data as if it were in XML format[3].

With this growing wealth of data available in XML format, it's becoming more and more important to find ways of getting meaningful data from what's available – of turning data into information. To this end, the W3C has created the XQuery Working Group with the goal of providing the tools to query both "real and virtual" sources of data on the World Wide Web[4]. While some experts[5] have voiced doubts about the direction of XML and its query languages, it's clear that more and more data is becoming available in XML format, and that access to this information is increasingly desirable.

XQuery and XSLT – the two most common, and most developed responses to this growing need – share a basic model of the documents (or data sources) on which they operate, using XPath to step through documents as if traversing a tree-like structure to return a sequence of appropriate items[6]. In a sense, they can be thought of as “discovering”, the items to return – generally nodes from the original document (or aggregates generated from those nodes).

SQL, which operates on traditional relational databases, on the other hand, is not generally conceived of as traversing anything. Relational databases are tuple-oriented, and (simplistically) can be seen as examining each tuple (or group of tuples) for inclusion in the answer.

The reason for these very different generalizations about the query languages lies on the differences between their underlying structures. XQuery, dealing with XML, operates on semi-structured data. Any given piece of information about a subject may or may not be present for any given instance. For example, if the color of a car is not known, the color tag will be missing. SQL, operating on (more or less) relational databases, operates on very structured data. If the database is capable of storing the color of a car, it will always have a field available for that purpose (whether or not it is used).

Perceived Problems with XQuery

Perhaps largely because of its dependence on XPath, XQuery requires a detailed knowledge of the structure of the document on which it operates, a structure that can be much more complex and varied than that of a SQL database. Since the fourth goal of XML is machine-readability[7], it should be no surprise that Drs Lakshmanan and Sadri seek to take advantage of this in their paper “On the Information Content of an XML Database”[8] by suggesting a method for simplifying the view of the data presented to users to the more familiar relational model and for enabling SQL queries upon this view to be carried out on the underlying XML data.

While some crossover research between relational databases and XML has been done (for example, [14]), the work seems to focus on transforming one data type into another for storage or transport, rather than enabling the users to take advantage of new technologies with existing skill sets.

Some other research has been done to simplify the querying of XML data sources, but the two thrusts of the research seem to be either adding functionality to XQuery or removing query languages from the equation. We will visit this research in Related Work.

CHAPTER II

PREVIOUS WORK

The foundation upon which I attempt to build is Drs Laskshmanan and Sadri's "On the Information Content of an XML Database"[8]. The core of this paper is the transformation of an XML document from a graph or tree structure into a table (or series of tables). In order to do this, any cycles are removed from the graph by renaming nodes, turning it into a tree. From there, each edge of the tree – each connection between nodes – is viewed as a binary table/relation with a pointer to the parent node in the left position and a pointer to the child in the right. Where an element might have contained another element of its own type, the transformation into tree would make this an element—subelement relation. If a full outer join of the tables is performed, then what Drs Lakshmanan and Sadri call an Information Content Tableau (ICT) is formed. An ICT is essentially a view of the XML document as a rather large table/relation. (Although some additional tables are used to represent certain relationships, such as IDREFS.)

Of course, this view need not be materialized. That is, the document need not be actually transformed into a table. Knowing its construction, they provide a method for translating SQL on the view into XQuery on the underlying document. The path information garnered in creating the ICT is stored so that any attributes used in a SQL query on the ICT can be translated to the XPath expressions needed to define their variables in the resulting XQuery.

Graph to Tree

The first step in this process is to view the schema of the XML document as a graph. This is the schema graph; its nodes are the elements of the document, and its edges are the parent-child relationships between the elements. Because this graph can contain cycles or because the same element may be a child of two different parents (the graph could be a directed acyclic graph), the cycles and repetitions must be smoothed out. This is done through renaming all nodes that participate in cycles. A node with questionable parentage is simply renamed to indicate its position more clearly. With these irritants removed, the graph is now a tree called the document tree.

Tree to Tables

Once the document tree has been created, it can be reconsidered as a series of binary tables/relations where the left hand side is the node identifier of the parent node and the right hand side the identifier of the child. Performing a full outer join of these binary tables/relations will produce a single table/relation containing all the information found in the original document, but flattened out. In the process of flattening the tree into this Hierarchical Information Content Tableau (HICT), a lot of repetition will occur: for each leaf node in the tree, a full row of its path data will be repeated in the table. However, this view need not be materialized.

In order to represent the relationships created by IDREF(S) and keyref mechanisms in XML, additional tables are sometimes created. These tables are called Linked Information Content Tableaus (LICT) and are created from the subtrees of these links. This work is focused on the HICT, however.

SQL to XQuery

Considering this new view of the XML document as a table (or relation), the user can now write SQL statements against it. To provide answers of more than just the appropriate node identifiers, the authors have included a value function that returns the value stored in a node. Here, we'll assume that function is implicit in the SQL.

All the attributes named in the SQL query must become variable declarations in the resulting XQuery. This is done by putting the nodes to which the attributes refer into a priority queue and replacing the lowest level node in the queue with its parent, attaching to the parent the path information from its child. This will obviously lead to some nodes being repeated in the priority queue as their children are processed. When this happens, the algorithm has found a Least Common Ancestor, which becomes a single variable declaration upon which the previous variables depend. The final node in the priority queue will be bound the XML document and will support the previously defined variables.

The core of the translation is handled in Algorithm 4.1, excerpted below:

For each entry (*n,l,p*) in the priority queue *P* that has the lowest level (largest *l*), replace (*n,l,p*) with (*parent, l* – 1*, n/p*), where *parent* is the parent of node *n* in the schema tree *T*. Note that *n/p* is the path formed by adding n to the beginning of the path *p*. The algorithm guarantees that the first node of path *p* is a child of *n*.

If a node *m* is listed more than once in the piority queue *P*, we have found a least common ancestor. Note that *m* must be at the lowest level (corresponding to level *l* – 1 in the previous step). (1) For each occurance (*m, k, p*) of a repeated node *m*, generate the declaration (for the decendent corresponding to path *p*) as follows: let *O* be the last node of the path *p*.

```
for $O in $M/p
```

Hence, all such variables as $O are declared using $M, the cariable corresponding to their least common ancestor *m*. The declaration for $M itself will be added at a later iteration.

(2) Replace all entries corresponding to a repeated node *m* (such as (*m, k, p*)) by a single entry (*m, k,* -) in the priority queue.

Repeat the previous steps as long as the priority queue has more than one entry.

When *P* has only one entry, it has to be of the form (*m, k,* -). Add the declaration

```
for $M in doc(…)//m
```

and remove the last entry from *P*.

Queries involving aggregation are treated somewhat differently. The use of let in XQuery variable declarations allows the query to range across a set of values. In section 4.1.5 of their paper, Lakshmanan and Sadri present Algorithm 4.4 for translating aggregate queries:

(1) Declaration for a variable $G corresponding to the group-by attribute is generated using `distinct-values` function.

(2) Declarations for variables corresponding to SQL query attributes are generated using a slightly modified version of Algorithm 4.1. Variables corresponding to aggregate attributes and their auxiliary variables should be declared using `let` (instead of `for`). Further, equality conditions to the group-by variable $G are included (see example 4.5)

(3) Finally, the XQuery return clause is generated by a simple translation for the SQL select clause.

I seek to flesh out and extend this algorithm to properly handle aggregate queries containing several group-by attributes. In addition, I seek to combine it and Algorithm 4.1 to produce a single, unified algorithm for translating SQL queries against HICT to XQuery on the base XML document.

CHAPTER III

CONTRIBUTIONS

My goal was to extend the aggregate query algorithm to handle multiple group-by attributes and flesh it out enough for implementation. To do this, I began by working carefully through the given algorithm for the standard case before moving on to aggregate queries.

Improvements to Algorithms: Algorithm 4.1 (Translation)

Data Structure

The algorithm as given suggests, for the ease of explanation, the use of a priority queue. A priority queue is a container class available through the Standard Template Library that effectively provides restricted access to a container, usually a vector[9]. A priority queue sorts its contents with the < operator to rank them, so that the lowest (or highest) ranked item is available, and no others. As given, the priority queue would sort on level.

However, since collision detection is necessary for the algorithm – that is, determining when several items in the data structure are not only on the same level, but also have the same node number, these tests would require popping and replacing nodes on top of the priority queue very frequently.

As such, I have worked with a vector of deques to represent the nodes upon which the algorithm is operating. Each deque holds all the nodes on a given level, so when the algorithm tests for collisions, it need only scan a single level, thereby avoiding the unnecessary work of popping and replacing elements on the priority queue.

At each progressive step through the algorithm, the lowest level (the last deque in the vector) is examined and if empty, removed.

I have also added the document to the data structure representing the HICT. This has simplified translation of queries (see below) and allows for future extensions for multiple documents (see Conclusions and Future Work).

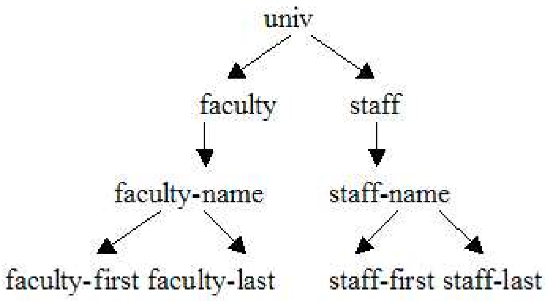
Ambiguity

In Algorithm 4.1 as given,

> When *P* has only one entry, it has to be of the form *(m, k, -)*. Add the declaration
> `for $M in doc(…)//m`
> and remove the last entry from *P*.

This may occasionally produce erroneous results. Consider the DTD below:

```
<!ELEMENT univ(faculty, staff)>
<!ELEMENT faculty(name*)>
<!ELEMENT staff(name*)>
<!ELEMENT name(first,
last)>
<!ELEMENT first #CDATA>
<!ELEMENT last #CDATA>
```



This will produce the tree to the right, which will translate into an HICT with the following attributes:

univ
faculty
faculty-name
faculty-first
faculty-last
staff
staff-name
staff-first
staff-last

So the SQL:

```
     SELECT faculty-first,
faculty-last
     FROM HICT
```

Create two entries in priority queue *P*:

(5, 3, -)
(6, 3, -)

Which will become:

(3, 2, first)
(3, 2, last)

Which will become:

(3, 2, -)
(Plus declarations for first and last name off this LCA.)

After this, I'll use the more compact notation below:

Node	**Level**	**Path**
Init:		
5	3	-
6	3	-
Step 1:		
3	2	first
3	2	last
Step 2:		
3	2	-
Plus declarations for first and last.		
Step 3:		
`for $name in doc(…)//name`		

At this point, the algorithm will add the declaration:

```
     for $name in doc(…)//name
```

because the XML element for this schema tree node is "name." So the final XQuery would be:

```
     for $name in doc(…)//name
     for $faculty-first in $name/first
     for $faculty-last  in $name/last
     return <row> {$faculty-first} {$faculty-last} </row>
```

However, in this case, the generated XQuery would be equivalent to the SQL:

```
SELECT faculty-first, faculty-last
FROM HICT
UNION ALL
SELECT staff-first, staff-last
FROM HICT
```

This is not intended.

The intuitive solution is to have the last node in *P* traverse all the way to the root node of the document before generating the declaration. So when *P* holds:

Continued from the previous example:

Node	Level	Path
Step 3:		
1	1	name
Step 4:		
0	0	faculty/name
Step 5:		
`for $name in doc(…)/univ/faculty/name`		

(3, 2, -)

It would become

(1, 1, name)

Then

(0, 0, faculty/name)

And then would produce the declaration:

```
for $name in doc(…)/univ/faculty/name
```

Which would produce the XQuery:

```
for $name in doc(…)/univ/faculty/name
for $faculty-first in $name/first
for $faculty-last  in $name/last
return <row> {$faculty-first} {$faculty-last} </row>
```

Which would be equivalent to the original SQL query.

SQL AS Keyword

An HICT query might look like

```
SELECT supplier-name, contact, count(part_id), sum(price)
FROM suppliersAlt.xml
GROUP BY supplier-name, contact;
```

And as such would be translated as

```
for $G1 in distinct-
values(doc("suppliersAlt.xml")/suppliers/supplier/name)
for $G2 in distinct-
values(doc("suppliersAlt.xml")/suppliers/supplier[name=$G1]
/contact)
let $alca :=   for  $temp0 in
doc("suppliersAlt.xml")//supplier
          where     $temp0/name = $G1 and
              $temp0/contact = $G2
          return    $temp0/part
let $CPid :=   $alca/part_id,
$SP :=         $alca/price
return   <row> {$G1} {$G2} {count($CPid)} {sum($SP)}</row>
```

Its results would resemble

```
<row>Sup1 contact@sup1.com 4 27.2</row>
<row>Sup2 sales@sup2.com 2 11.88</row>
<row>Sup3 bob@sup3.co.uk 2 22</row>
```

This could be difficult to parse if the items contained spaces. I.E. “Sup 1” instead of

“Sup1”. One solution would be to make the return statement look like this:

```
return <row>”{$G1}”,”{$G2}”,”{count($CPid)}
”,”{sum($SP)}”</row>
```

The output here would be

```
<row>”Sup1”,”contact@sup1.com”,”4”,”27.2”</row>
<row>”Sup2”,”sales@sup2.com”,”2”,”11.88”</row>
<row>”Sup3”,”bob@sup3.co.uk”,”2”,”22”</row>
```

This would be more similar to a CSV file. Or the output could be structured like:

```
     return <row>
     <supplier-name>{$G1}</supplier-name >
     <contact>{$G2}</contact>
     <countpart>{count($CPid)}</countpart>
     <sumprice>{sum($SP)}</sumprice>
 </row>
```

Or

```
     return <row>
          {element supplier-name {$G1}}
          {element contact {$G2}}
          {element countpart {count($CPid)}}
          {element sumprice {sum($SP)}}
      </row>
```

Either of these would create output like:

```
<row>
   <supplier-name>Sup1</supplier-name>
   <contact>contact@sup1.com</contact>
   <countpart>4</countpart>
   <sumprice>27.2</sumprice>
</row>
<row>
   <supplier-name>Sup2</supplier-name>
   <contact>sales@sup2.com</contact>
   <countpart>2</countpart>
   <sumprice>11.88</sumprice>
</row>
<row>
   <supplier-name>Sup3</supplier-name>
   <contact>bob@sup3.co.uk</contact>
   <countpart>2</countpart>
   <sumprice>22</sumprice>
</row>
```

This output has the advantage of clearly defining field names and clearly separating variables. It also makes the output as easily parsed for XML use as for import into a relational database.

The node names could be generated automatically with the value of the node/attribute used when available, and a random string generated when necessary (as MS Access does for aggregate results), or could be specified by the SQL keyword `AS`.

```
SELECT supplier-name, contact, count(part_id) AS countpart,
sum(price) AS sumprice
FROM suppliers
GROUP BY supplier-name, contact;
```

And as such would be translated as the above examples. As it seems to put fewer demands on the XQuery engine executing the query, I have opted to use the former method.

Aggregate Queries

At this point, we are more or less familiar with the translation algorithms 4.1 and 4.4. The unified algorithm is given on the following page, but let’s look at some of the points of note. The original algorithms only tracked 3 pieces of information about each node – node id, level, and path. In order to properly handle the broader range of queries, this algorithm tracks the 3 items above, as well as node type (group-by or non-group-by), intended output, and variable name. Also tracked for each node, but not listed below, are the nodes with which it has collided. This last item is necessary to ensure that group-by nodes are not repeatedly colliding.

The significant changes to algorithms 4.1 and 4.4 (that is, those not mentioned above) occur at the collisions between nodes. When collide, it is important to note their type and the nodes with which they have collided before. That is what enables the algorithm to determine how to handle each collision.

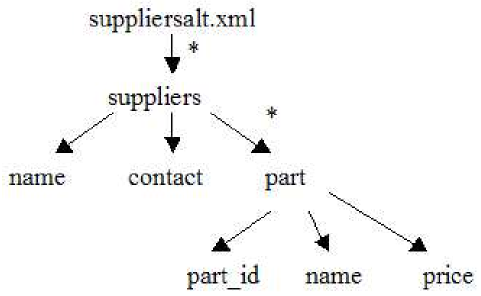
Following the listing of the algorithm are three examples that offer step by step walks through the algorithm’s handling of several different situations. These are intended to supplement the examples provided in “On the Information Content of an XML Database,” not to replace them.

Unified Algorithm for Translating Queries on HICT

1 Declarations for variables corresponding to SQL query attributes are generated using a slightly modified version of Algorithm 4.1. Variables corresponding to aggregate attributes and their auxiliary variables should be declared using let (instead of for) and should use the distinct-values function. Further, equality conditions to the group-by variables $G1..n are included. If one or more group-by variables exist, all non-aggregate, non-group-by variables should be handled in an embedded FLWOR statement, created with this algorithm, whose where condition should contain bindings to the group-by attributes. The variable resulting from this embedded FLWOR should be placed in the result statement of the outer FLWOR.

   a. Feed all attributes into a vector of deques P in the form (n,l,p,t,o,v) in the form (node id, level, path, type, output, variable name). Group-by variables are marked as type 0, others as type 1. Output will be initialized to the original XML node name, and is used for the LCA of non-group-by variables. The variable name should be given based on an AS statement or the attribute name.
   b. Proceed as Algorithm 4.1, with these modifications:
      i. When nodes of type 0 duplicate each other – when they collide:
         1. To the second (and later) of the set of duplicates add to the front of its path
            a. Current node name
            b. Predicate(s) modifying current node name for each previous node in set of duplicates
               i. Path of previous duplicate
               ii. =
               iii. Variable name of previous duplicate
         2. Mark these nodes so that they will not collide again.
      ii. When nodes of type >=1 have collided, sum their types and proceed as Alg 4.1
         1. The output slot is filled with the top of the path of the replacement node so that the result of the embedded FLWOR is the LCA.
      iii. When nodes of type 0 collide with non-group-by nodes (of type >=1), create the where clause (for the embedded FLOWR) for each node of type 0 such that:
         1. [path of node] = [var of type 0 node]

Example 1



Consider SuppliersAlt.xml described by the graph to the right. The document is available in the appendix. Its ICT has the following attributes:

doc("suppliersalt.dtd")
suppliers
supplier
supplier-name (LEAF)
contact (LEAF)
part
part_id (LEAF)
price (LEAF)
part-name (LEAF)

To translate the following query on this document

```
SELECT supplier-name, contact, count(part), sum(price)
FROM suppliersalt.xml
GROUP BY supplier-name, contact
```

Step 1:

(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,1,part,$countpart)
(5: 7,4, ,1,price,$sumprice)

In this first step, the attributes from the query are converted into nodes for the translation process.

Step 2:

```
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,1,part,$countpart)
(5: 5,3,price,1,price,$sumprice)
----------
```

Here, the node with the lowest level has been advanced to its parent node, and its path has been updated.

Step 3:

```
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,1,part,$countpart)
(5: 5,3,price,1,price,$sumprice)
----------
```

Here, the lowest level of the translation data structure is empty, so it has been removed.

Step 4:

```
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,2,part,$RANDB)
----------
```

Nodes 4 and 5 have collided, so they are replaced with a single node. The previous nodes are written to output as:

```
    let $sumprice := $RANDB/price
    let $countpart := $RANDB
```

Both are bound to this new node, which is instructed to bind its output to the part node which these dependent variables expect.

```
Step 5:
----------
(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,2,part,$RANDB)
----------
```
Since all the nodes were on the same level, but did not collide, the first node was advanced.
```
---------------------------------------
Step 6:
----------
(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 2,2,contact,0, ,$contact)
(4: 5,3, ,2,part,$RANDB)
----------
```
Node 3 was advanced because it was the first node on the lowest level of the data structure.
```
---------------------------------------
Step 7:
----------
(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 2,2,contact,0, ,$contact)
(4: 2,2,part,2,part,$RANDB)
----------
```
All nodes have now been advanced to level 2, where they shared a parent.
```
---------------------------------------
```

Step 8:

(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 2,2,contact,0, ,$contact)
(4: 2,2,part,2,part,$RANDB)

Here, the lowest level of the translation data structure is empty, so it has been removed.

Step 9:

(uid: n,l,p,t,o,v)
(2: 1,1,supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 1,1,supplier/contact,0, ,$contact)

All the nodes present were in collision, so the group-by nodes (type 0) modified each others' paths to ensure they would bind to the same data instance, and the non-group by node (node 4) was bound to an embedded FLWOR statement matching those group by elements:

```
     let $RANDB := for $temp in
     doc("suppliersalt.xml")/suppliers/supplier
     where $temp/name = $supplier-name
     and $temp/contact = $contact
     return $temp/part
```

Step 10:

(uid: n,l,p,t,o,v)
(2: 1,1,supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 1,1,supplier/contact,0, ,$contact)

The group-by nodes, having previously collided, are marked to avoid future collisions.

Step 11:

(uid: n,l,p,t,o,v)
(2: 0,0,suppliers/supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 1,1,supplier/contact,0, ,$contact)

So they are advanced each in turn, until they reach the root of the document.

Step 12:

(uid: n,l,p,t,o,v)
(2: 0,0,suppliers/supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 0,0,suppliers/supplier/contact,0, ,$contact)

Step 13:

(uid: n,l,p,t,o,v)
(2: 0,0,suppliers/supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 0,0,suppliers/supplier/contact,0, ,$contact)

Step 14:

(uid: n,l,p,t,o,v)
(3: 0,0,suppliers/supplier/contact,0, ,$contact)

Here, the first (node 2) has reached the root, and so has been bound to its output statement:

```
     for $supplier-name in distinct-
values(doc("suppliersalt.xml")/suppliers/supplier
```

Step 15:

(uid: n,l,p,t,o,v)

Here, the second has reached the root and been bound to output:
```
     for $contact in distinct-
values(doc("suppliersalt.xml")/suppliers/supplier/contact)
```

The translation structure now being empty, the XQuery written so far is:
```
for $contact in distinct-
values(doc("suppliersalt.xml")/suppliers/supplier/contact)
for $supplier-name in distinct-
values(doc("suppliersalt.xml")/suppliers/supplier[contact =
$contact]/name)
let $RANDB := for $temp in
doc("suppliersalt.xml")/suppliers/supplier
where $temp/name = $supplier-name
and $temp/contact = $contact
return $temp/part

let $sumprice := $RANDB/price
let $countpart := $RANDB
```

From here, we create the return statement as a mirror of the SQL query's SELECT clause:

```
return <row>
     <supplier-name>{$supplier-name}</supplier-name>
     <contact>{$contact}</contact>
     <countpart>{count($countpart)}</countpart>
     <sumprice>{sum($sumprice)}</sumprice>
     </row>
```

This produces our final XQuery:

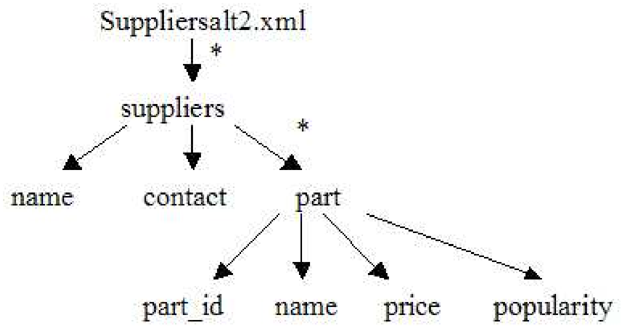
```
for $contact in distinct-
values(doc("suppliersalt.xml")/suppliers/supplier/contact)
for $supplier-name in distinct-
values(doc("suppliersalt.xml")/suppliers/supplier[contact =
$contact]/name)
let $RANDB := for $temp in
doc("suppliersalt.xml")/suppliers/supplier
where $temp/name = $supplier-name
and $temp/contact = $contact
return $temp/part

let $sumprice := $RANDB/price
let $countpart := $RANDB
return <row>
     <supplier-name>{$supplier-name}</supplier-name>
     <contact>{$contact}</contact>
     <countpart>{count($countpart)}</countpart>
     <sumprice>{sum($sumprice)}</sumprice>
     </row>
```

Example 2

Consider SuppliersAlt2.xml described by the graph below right. The document is available in the appendix. Its ICT has the following attributes:

doc("suppliersalt2.dtd")
suppliers
supplier
supplier-name (LEAF)
contact (LEAF)
part
part_id (LEAF)
price (LEAF)
part-name (LEAF)
popularity (LEAF)



To translate the following query on this document:

```
SELECT supplier-name, contact, count(part), sum(price),
avg(popularity)
FROM suppliersalt2.xml
GROUP BY supplier-name, contact
```

Step 1:

(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,1,part,$countpart)
(5: 7,4, ,1,price,$sumprice)
(6: 9,4, ,1,popularity,$avgpopularity)

In this first step, the attributes from the query are converted into nodes for the translation process.

```
Step 2:
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,1,part,$countpart)
(5: 5,3,price,1,price,$sumprice)
(6: 9,4, ,1,popularity,$avgpopularity)
----------
The first node on the lowest level is advanced.
----------------------------------------
Step 3:
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,1,part,$countpart)
(5: 5,3,price,1,price,$sumprice)
(6: 5,3,popularity,1,popularity,$avgpopularity)
----------
The first node on the lowest level is advanced.
----------------------------------------
Step 4:
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,1,part,$countpart)
(5: 5,3,price,1,price,$sumprice)
(6: 5,3,popularity,1,popularity,$avgpopularity)
----------
The lowest level, being empty, is removed from our data structure.
----------------------------------------
```

Step 5:

(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,3,part,$RANDB)

Nodes 4, 5 and 6 have collided and are replaced by a single node bound to their point of collision. Output for the removed nodes:

```
     let $avgpopularity := $RANDB/popularity
     let $sumprice := $RANDB/price
     let $countpart := $RANDB
```

Step 6:

(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 4,3, ,0, ,$contact)
(4: 5,3, ,3,part,$RANDB)

The first node on the lowest level is advanced.

Step 7:

(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 2,2,contact,0, ,$contact)
(4: 5,3, ,3,part,$RANDB)

The first node on the lowest level is advanced.

Step 8:

(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 2,2,contact,0, ,$contact)
(4: 2,2,part,3,part,$RANDB)

The first node on the lowest level is advanced.

Step 9:

(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 2,2,contact,0, ,$contact)
(4: 2,2,part,3,part,$RANDB)

The lowest level, being empty, is removed from our data structure.

Step 10:

(uid: n,l,p,t,o,v)
(2: 1,1,supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 1,1,supplier/contact,0, ,$contact)

All the nodes present were in collision, so the group-by nodes (type 0) modified each others' paths to ensure they would bind to the same data instance, and the non-group by node (node 4) was bound to an embedded FLWOR statement matching those group by elements:

```
     let $RANDB := for $temp in
doc("suppliersalt2.xml")/suppliers/supplier
     where $temp/name = $supplier-name
     and $temp/contact = $contact
     return $temp/part
```

Step 11:

(uid: n,l,p,t,o,v)
(2: 1,1,supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 1,1,supplier/contact,0, ,$contact)

The group-by nodes, having previously collided, are marked to avoid future collisions.

Step 12:

(uid: n,l,p,t,o,v)
(2: 0,0,suppliers/supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 1,1,supplier/contact,0, ,$contact)

The first node on the lowest level is advanced.

Step 13:

(uid: n,l,p,t,o,v)
(2: 0,0,suppliers/supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 0,0,suppliers/supplier/contact,0, ,$contact)

The first node on the lowest level is advanced.

Step 14:

(uid: n,l,p,t,o,v)
(2: 0,0,suppliers/supplier[contact = $contact]/name,0, ,$supplier-name)
(3: 0,0,suppliers/supplier/contact,0, ,$contact)

The lowest level, being empty, is removed from our data structure.

Step 15:

(uid: n,l,p,t,o,v)
(3: 0,0,suppliers/supplier/contact,0, ,$contact)

Node 2, having reached the root, is removed and bound to its output statement:
```
     for $supplier-name in distinct-
values(doc("suppliersalt2.xml")/suppliers/supplier[contact
= $contact]/name)
```

Step 16:

(uid: n,l,p,t,o,v)

Node 3, having reached the root, is removed and bound to its output statement:

```
     for $contact in distinct-
values(doc("suppliersalt2.xml")/suppliers/supplier/contact)
```

Since the translation structure is empty, we can now add our return statement to the previous output to complete our XQuery:

```
for $contact in distinct-
values(doc("suppliersalt2.xml")/suppliers/supplier/contact)
for $supplier-name in distinct-
values(doc("suppliersalt2.xml")/suppliers/supplier[contact
= $contact]/name)
let $RANDB := for $temp in
doc("suppliersalt2.xml")/suppliers/supplier
where $temp/name = $supplier-name
and $temp/contact = $contact
return $temp/part
let $avgpopularity := $RANDB/popularity
let $sumprice := $RANDB/price
let $countpart := $RANDB
return <row>
     <supplier-name>{$supplier-name}</supplier-name>
     <contact>{$contact}</contact>
     <countpart>{count($countpart)}</countpart>
     <sumprice>{sum($sumprice)}</sumprice>
     <avgpopularity>{avg($avgpopularity)}</avgpopularity>
     </row>
```

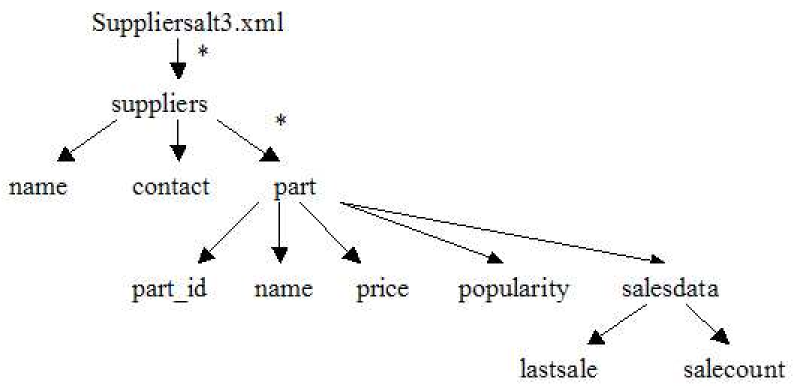
Example 3

To demonstrate the algorithm's ability to handle multiple LCAs between the various attributes, consider the following SQL query:

```
     SELECT sname, count(part_id), sum(salecount),
max(lastsale)
     FROM HICT(suppliersAlt3.xml)
     GROUP BY sname
```

SuppliersAlt3.xml has the following schema tree. Its ICT has the following attributes:

doc("suppliersalt3.dtd")
suppliers
supplier
supplier-name (LEAF)
contact (LEAF)
part
part_id (LEAF)
price (LEAF)
part-name (LEAF)
popularity (LEAF)
salesdata
lastsale (LEAF)
salecount (LEAF)



To translate the following query on this document:

```
SELECT supplier-name, count(part_id), sum(salecount),
max(lastsale)
FROM suppliersalt3.xml
GROUP BY supplier-name
```

```
Step 1:
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 6,4, ,1,part_id,$countpart_id)
(5: 11,5, ,1,lastsale,$maxlastsale)
(4: 12,5, ,1,salecount,$sumsalecount)
----------
```
In this first step, the attributes from the query are converted into nodes for the translation process.
```
---------------------------------------
Step 2:
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 6,4, ,1,part_id,$countpart_id)
(5: 10,4,lastsale,1,lastsale,$maxlastsale)
(4: 12,5, ,1,salecount,$sumsalecount)
----------
```
The first node on the lowest level is advanced.
```
---------------------------------------
Step 3:
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 6,4, ,1,part_id,$countpart_id)
(4: 10,4,salecount,1,salecount,$sumsalecount)
(5: 10,4,lastsale,1,lastsale,$maxlastsale)
----------
```
The first node on the lowest level is advanced.
```
---------------------------------------
Step 4:
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 6,4, ,1,part_id,$countpart_id)
(4: 10,4,salecount,1,salecount,$sumsalecount)
(5: 10,4,lastsale,1,lastsale,$maxlastsale)
----------
```
The lowest level, being empty, is removed from our data structure.
```
---------------------------------------
```

Step 5:
```
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 6,4, ,1,part_id,$countpart_id)
(4: 10,4, ,2,salesdata,$RANDB)
----------
```
Nodes 4 and 5 have collided and are replaced by a single node bound to their point of collision. Output for the removed nodes:
```
      let $maxlastsale := $RANDB/lastsale
      let $sumsalecount := $RANDB/salecount
```

Step 6:
```
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 5,3,part_id,1,part_id,$countpart_id)
(4: 10,4, ,2,salesdata,$RANDB)
----------
```
The first node on the lowest level is advanced.

Step 7:
```
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 5,3,part_id,1,part_id,$countpart_id)
(4: 5,3,salesdata,2,salesdata,$RANDB)
----------
```
The first node on the lowest level is advanced.

Step 8:
```
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 5,3,part_id,1,part_id,$countpart_id)
(4: 5,3,salesdata,2,salesdata,$RANDB)
----------
```
The lowest level, being empty, is removed from our data structure.

Step 9:
```
----------
(uid: n,l,p,t,o,v)
(2: 3,3, ,0, ,$supplier-name)
(3: 5,3, ,2,part,$RANDC)
----------
```
Nodes 3 and 4 have collided, so they are replaced by a single node to which their outputs are bound:
```
     let $RANDB := $RANDC/salesdata
     let $countpart_id := $RANDC/part_id
```

Step 10:
```
----------
(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 5,3, ,2,part,$RANDC)
----------
```
The first node on the lowest level is advanced.

Step 11:
```
----------
(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 2,2,part,2,part,$RANDC)
----------
```
The first node on the lowest level is advanced.

Step 12:
```
----------
(uid: n,l,p,t,o,v)
(2: 2,2,name,0, ,$supplier-name)
(3: 2,2,part,2,part,$RANDC)
----------
```
The lowest level, being empty, is removed from our data structure.

Step 13:

(uid: n,l,p,t,o,v)
(2: 1,1,supplier/name,0, ,$supplier-name)

Node 3 has collided with the group-by node (node 2), and is so removed and bound to an embedded FLWOR statement:

```
      let $RANDC := for $temp in
      doc("suppliersalt3.xml")/suppliers/supplier
      where $temp/name = $supplier-name
      return $temp/part
```

The group-by node (node 2) is advanced.

Step 14:

(uid: n,l,p,t,o,v)
(2: 1,1,supplier/name,0, ,$supplier-name)

The lowest level, being empty, is removed from our data structure.

Step 15:

(uid: n,l,p,t,o,v)
(2: 0,0,suppliers/supplier/name,0, ,$supplier-name)

The first node on the lowest level, in this case, the only node, is advanced.

Step 16:

(uid: n,l,p,t,o,v)
(2: 0,0,suppliers/supplier/name,0, ,$supplier-name)

The lowest level, being empty, is removed from our data structure.

Step 17:

(uid: n,l,p,t,o,v)

Node 2 has reached the root and is bound to its output statement:
```
     for $supplier-name in distinct-
values(doc("suppliersalt3.xml")/suppliers/supplier/name)
```

Since the translation structure is empty, we can now add our return statement to the previous output to complete our XQuery:

```
for $supplier-name in distinct-
values(doc("suppliersalt3.xml")/suppliers/supplier/name)
let $RANDC := for $temp in
doc("suppliersalt3.xml")/suppliers/supplier
where $temp/name = $supplier-name
return $temp/part

let $RANDB := $RANDC/salesdata
let $countpart_id := $RANDC/part_id
let $maxlastsale := $RANDB/lastsale
let $sumsalecount := $RANDB/salecount
return <row>
     <supplier-name>{$supplier-name}</supplier-name>
     <countpart_id>{count($countpart_id)}</countpart_id>
     <sumsalecount>{sum($sumsalecount)}</sumsalecount>
     <maxlastsale>{max($maxlastsale)}</maxlastsale>
     </row>
```

CHAPTER IV

RELATED WORK

Schema-Free XQuery

In this paper[10], the authors offer an extension to XQuery intend to reduce its complexity. Their Meaningful Lowest Common Ancestor Structure (MCLAS) attempts to automatically determine a meaningful relationship between nodes by extending the concept of lowest (or least) common ancestor to test the type of the intervening nodes to avoid duplication.

XSearch: A Semantic Search Engine for XML

XSearch[11] attempts to remove the query language from searching XML documents. It instead implements a search engine-like interface wherein the user specifies both node content and node labels to match. It tests that nodes are meaningfully related by comparing labels in the document tree as does “Schema-Free XQuery” above.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

While many options are available for querying XML databases, HICT provides an alternative familiar to a broad range of users. With the inclusion of a greater range of aggregate queries, HICT becomes much more useful. Some questions and further work remain: integrating LICT with aggregate queries, handling NULLS, and queries against multiple documents present themselves as pressing problems for further work. I present some quick notes on the last two of these items below.

This version and its implementation use a very rudimentary output system. For more control over output, a system like SQL/XML should be used.

Nulls and Missing Data – Two- vs. Three-Way Logic

Michael Kay (the creator of SAXON) points out in *XQuery from the Experts*[12] (p112-7) that SQL evaluates statements with three values: true, false, and unknown (or null), but XQuery's basic comparison operators use Boolean logic, treating unknowns or missing elements as false, so that

```
//car[mileage <= 25]
```

would not return cars with an unknown mileage value (unknown compared to 25 is false), but

```
//car[not(mileage > 25)]
```

would, because the unknown value would return false when compared against 25, and that expression is negated. SQL, on the other hand, would return the unknown for both, and wouldn't include cars with null mileage in the second query.

He proposes three functions (Listing 2.2, p 116) to provide SQL-like functionality to XQuery's `equal`, `not`, `and`, and `or` functions by mapping the SQL truth tables against these functions.

He also makes the point that XQuery has several ways of handling unknown data – a missing or an empty element tag. Whenever XQuery operates on missing data, it returns an empty sequence, which of course evaluates to false in the above logic.

This raises some interesting questions about `IS NULL` in SQL on ICT. Further investigation is necessary to see if this could be solved by implementing Kay's functions for many of the operators and translating IS NULL to a test to see if a sequence is empty, but this would affect the transitivity of XQuery's operators.

Multi-document Queries

Since XML is a widely used format for data interchange, it is to be expected that a user might want to query more than one document at a time. In XQuery, this is achieved by binding variables to different documents. For example:

```
for    $a in doc("doc1.xml")//author,
       $b in doc("doc2.xml")//author
return <authors>{($a, $b)}</authors>
```

This query would return the author nodes in doc1.xml, followed by the authors in doc2.xml. This result is (ignoring document order) similar to the SQL statement:

```
SELECT author FROM table1
UNION ALL
SELECT author FROM table2
```

When two documents are used in the same query, there are two cases into which the join can fall: 1) The two documents are similar in structure, and can be viewed as multiple instances of the same document (Two departments in a university publish HR information in XML format) or 2) The two documents are dissimilar in structure, but are joined by some foreign key. (A corporation publishes sales information in one XML document, and another department publishes supplier information in another; the parts sold by salesman and bought from suppliers can link the two).

If we wished to query across two documents with the same structure, our XQuery could be simply modified to include:

```
for   $D in for   $D1 in doc(doc1.xml),
                  $D2 in doc(doc2.xml)
              return <newroot>{($D1, $D2)}</newroot>
```

Because the two documents share the same DTD, the only necessary change to variable declaration would be to bind the variables to $D, and to bind $D to the subquery combining the documents. This would union the two documents.

When the two documents are dissimilar, however, the join between them requires more attention. In many cases, the join condition will not be placed where expected. Outer joins between documents, for example, may require embedding a FLWOR to handle one document inside the return statement of the statement handling the other.

REFERENCES

[1] Extensible Markup Language (XML). http://w3c.org/XML/

[2] About the World Wide Web Consortium (W3C). http://w3c.org/Consortium/#mission

[3] Microsoft SQL Server Rich XML Support http://www.microsoft.com/sql/evaluation/features/RichXML.asp; About Oracle XML Products http://www.oracle.com/technology/tech/xml/info/htdocs/otnwp/ about_oracle_xml_products.htm#9iFamily; and DB2 Universal Database for Linux, UNIX, and Windows http://www-306.ibm.com/software/data/db2/udb/

[4] W3C XML Query (XQuery) http://w3c.org/XML/Query

[5] Pascal, Fabian, "If You Liked SQL, You'll Love XQuery." www.dbazine.com/pascal19.shtml

[6] XQuery 1.0 and XPath 2.0 Data Model. http://w3c.org/TR/xpath-datamodel/

[7] Extensible Markup Language (XML) 1.0 (Third Edition). http://w3c.org/TR/2004/REC-xml-20040204/#sec-origin-goals

[8] Lakshmanan, Laks and Fereidoon Sadri. "On the Information Content of an XML Database". Unpublished.

[9] Ford, William and William Topp, Data Structures with C++ using STL, Second Edition. New Jersey: Prentice Hall, 2002.

[10] Li, Yunyao, Cong Yu and H.V. Jagadish "Schema-Free XQuery." Proceedings of the 30th VLDB Conference. Toronto, Canada, 2004

[11] Cohen, Sara, Jonathan Mamou, Yaron Kanza and Yehoshua Sagiv. "XSearch: A Semantic Search Engine for XML." Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003.

[12] Katz, Howard, Ed.. XQuery from the Experts. Boston: Addison-Wesley 2004.

[13] Brundage, Michael. XQuery: The XML Query Language. Boston: Addison-Wesley 2004.

[14] Lee, Dongwon, and Murali Mani and Wesley W. Chu. "Schema Conversion Methods between XML and Relational Models."

APPENDIX A: SUPPLIERSALT.XML

```
<suppliers>
   <supplier>
      <name>Sup1</name>
      <contact>contact@sup1.com</contact>
      <part>
         <part_id>1</part_id>
         <price>3.5</price>
         <name>Basic Widget</name>
      </part>
      <part>
         <part_id>5</part_id>
         <price>10</price>
         <name>X-treme Widget</name>
      </part>
   </supplier>
   <supplier>
      <name>Sup2</name>
      <contact>sales@sup2.com</contact>
      <part>
         <part_id>2</part_id>
         <price>5</price>
         <name>Basic Widget Plus Wedges</name>
      </part>
      <part>
         <part_id>4</part_id>
         <price>6.88</price>
         <name>Mid-rage Widget</name>
      </part>
   </supplier>
   <supplier>
      <name>Sup3</name>
      <contact>bob@sup3.co.uk</contact>
      <part>
         <part_id>3</part_id>
         <price>10</price>
         <name>Her Majesty's Widget</name>
      </part>
      <part>
         <part_id>6</part_id>
         <price>12</price>
         <name>His Majesty's Widget</name>
      </part>
   </supplier>
   <supplier>
      <name>Sup1</name>
      <contact>contact@sup1.com</contact>
      <part>
         <part_id>100</part_id>
         <price>3.6</price>
         <name>Basic Widget With Color</name>
      </part>
      <part>
         <part_id>500</part_id>
         <price>10.1</price>
         <name>X-treme Widget With Color</name>
      </part>
   </supplier>
</suppliers>
```

Its DTD would be equivalent to:

```
<!ELEMENT suppliers (supplier*)>
<!ELEMENT supplier (name, contact, part*)>
<!ELEMENT part (part_id, price, name)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT contact (#PCDATA)>
<!ELEMENT part_id (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

APPENDIX B: SUPPLIERSALT2.XML

```
<suppliers>
      <supplier>
            <name>Sup1</name>
            <contact>contact@sup1.com</contact>
            <part>
                  <part_id>1</part_id>
                  <price>3.5</price>
                  <name>Basic Widget</name>
                  <popularity>7</popularity>
            </part>
            <part>
                  <part_id>5</part_id>
                  <price>10</price>
                  <name>X-treme Widget</name>
                  <popularity>6</popularity>
            </part>
      </supplier>
      <supplier>
            <name>Sup2</name>
            <contact>sales@sup2.com</contact>
            <part>
                  <part_id>2</part_id>
                  <price>5</price>
                  <name>Basic Widget Plus Wedges</name>
                  <popularity>6.5</popularity>
            </part>
            <part>
                  <part_id>4</part_id>
                  <price>6.88</price>
                  <name>Mid-rage Widget</name>
                  <popularity>4</popularity>
            </part>
      </supplier>
      <supplier>
            <name>Sup3</name>
            <contact>bob@sup3.co.uk</contact>
            <part>
                  <part_id>3</part_id>
                  <price>10</price>
                  <name>Her Majesty's Widget</name>
                  <popularity>3</popularity>
            </part>
            <part>
                  <part_id>6</part_id>
                  <price>12</price>
                  <name>His Majesty's Widget</name>
                  <popularity>3.5</popularity>
            </part>
      </supplier>
      <supplier>
            <name>Sup1</name>
            <contact>contact@sup1.com</contact>
            <part>
                  <part_id>100</part_id>
                  <price>3.6</price>
                  <name>Basic Widget With Color</name>
                  <popularity>7.1</popularity>
            </part>
            <part>
                  <part_id>500</part_id>
                  <price>10.1</price>
                  <name>X-treme Widget With Color</name>
                  <popularity>4.2</popularity>
            </part>
      </supplier>
</suppliers>
```

Its DTD would be equivalent to:

```
<!ELEMENT suppliers (supplier*)>
<!ELEMENT supplier (name, contact, part*)>
<!ELEMENT part (part_id, price, name, popularity)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT contact (#PCDATA)>
<!ELEMENT part_id (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT popularity (#PCDATA)>
```

APPENDIX C: SUPPLIERSALT3.XML

```
<suppliers>
      <supplier>
            <name>Sup1</name>
            <contact>contact@sup1.com</contact>
            <part>
                  <part_id>1</part_id>
                  <price>3.5</price>
                  <name>Basic Widget</name>
                  <popularity>7</popularity>
                  <salesdata>
                        <lastsale>0410</lastsale>
                        <salecount>100</salecount>
                  </salesdata>
            </part>
            <part>
                  <part_id>5</part_id>
                  <price>10</price>
                  <name>X-treme Widget</name>
                  <popularity>6</popularity>
                  <salesdata>
                        <lastsale>0409</lastsale>
                        <salecount>90</salecount>
                  </salesdata>
            </part>
      </supplier>
      <supplier>
            <name>Sup2</name>
            <contact>sales@sup2.com</contact>
            <part>
                  <part_id>2</part_id>
                  <price>5</price>
                  <name>Basic Widget Plus Wedges</name>
                  <popularity>6.5</popularity>
                  <salesdata>
                        <lastsale>0409</lastsale>
                        <salecount>95</salecount>
                  </salesdata>
            </part>
            <part>
                  <part_id>4</part_id>
                  <price>6.88</price>
                  <name>Mid-rage Widget</name>
                  <popularity>4</popularity>
                  <salesdata>
                        <lastsale>0402</lastsale>
                        <salecount>50</salecount>
                  </salesdata>
            </part>
      </supplier>
```

```
<supplier>
            <name>Sup3</name>
            <contact>bob@sup3.co.uk</contact>
            <part>
                  <part_id>3</part_id>
                  <price>10</price>
                  <name>Her Majesty's Widget</name>
                  <popularity>3</popularity>
                  <salesdata>
                        <lastsale>0402</lastsale>
                        <salecount>30</salecount>
                  </salesdata>
            </part>
            <part>
                  <part_id>6</part_id>
                  <price>12</price>
                  <name>His Majesty's Widget</name>
                  <popularity>3.5</popularity>
                  <salesdata>
                        <lastsale>0312</lastsale>
                        <salecount>50</salecount>
                  </salesdata>
            </part>
      </supplier>
      <supplier>
            <name>Sup1</name>
            <contact>contact@sup1.com</contact>
            <part>
                  <part_id>100</part_id>
                  <price>3.6</price>
                  <name>Basic Widget With Color</name>
                  <popularity>7.1</popularity>
                  <salesdata>
                        <lastsale>0408</lastsale>
                        <salecount>80</salecount>
                  </salesdata>
            </part>
            <part>
                  <part_id>500</part_id>
                  <price>10.1</price>
                  <name>X-treme Widget With Color</name>
                  <popularity>4.2</popularity>
                  <salesdata>
                        <lastsale>0402</lastsale>
                        <salecount>65</salecount>
                  </salesdata>
            </part>
      </supplier>
</suppliers>
```

Its DTD would be equivalent to:

```
<!ELEMENT suppliers (supplier*)>
<!ELEMENT supplier (name, contact, part*)>
<!ELEMENT part (part_id, price, name, popularity, salesdata)>
<!ELEMENT salesdata(lastsale, salecount)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT contact (#PCDATA)>
<!ELEMENT part_id (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT popularity (#PCDATA)>
<!ELEMENT lastsale (#PCDATA)>
<!ELEMENT salecount (#PCDATA)>
```