Efficiently Publishing Relational Data as XML Documents

JAYAVEL SHANMUGASUNDARAM[1] EUGENE SHEKITA RIMON BARR[2] MICHAEL CAREY[3]

BRUCE LINDSAY HAMID PIRAHESH BERTHOLD REINWALD

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

{shanmuga, shekita}@almaden.ibm.com, barr@cs.cornell.edu, carey@propel.com
{bgl, pirahesh, reinwald}@almaden.ibm.com

Abstract: XML is rapidly emerging as a standard for exchanging business data on the World Wide Web. For the foreseeable future, however, most business data will continue to be stored in relational database systems. Consequently, if XML is to fulfill its potential, some mechanism is needed to publish relational data as XML documents. Towards that goal, one of the major challenges is finding a way to efficiently structure and tag data from one or more tables as a hierarchical XML document. Different alternatives are possible depending on when this processing takes place and how much of it is done inside the relational engine. In this paper, we characterize and study the performance of these alternatives. Among other things, we explore the use of new scalar and aggregate functions in SQL for constructing complex XML documents directly in the relational engine. We also explore different execution plans for generating the content of an XML document. The results of an experimental study show that constructing XML documents inside the relational engine can have a significant performance benefit. Our results also show the superiority of having the relational engine use what we call an "outer union plan" to generate the content of an XML document.

Keywords: Relational databases, XML, publishing

1. Introduction

XML is rapidly emerging as a standard for exchanging business data on the World Wide Web. Its nested, self-describing structure provides a simple yet flexible means for applications to exchange data. In fact, there are already many industry proposals [5] to standardize Document Type Descriptors (DTDs) [26], which are essentially schemas for XML documents. These DTDs are being developed for domains as diverse as electronic commerce [4] and real estate [18]. Despite the excitement surrounding XML, it is important to note that most operational business data, even for new web-based applications, continues to be stored in relational database systems. This is unlikely to change in the foreseeable future because of the reliability, scalability, tools, and performance associated with relational database systems. Consequently, if XML is to fulfill its potential, some mechanism is needed to publish relational data in the form of XML documents.

---

[1] Also at the University of Wisconsin, Madison, WI 53706.
[2] Work done while the author was visiting from Cornell University, Ithaca, NY 14850.
[3] Currently at Propel, 1010 Rincon Circle, San Jose, CA 95131.

There are two main requirements for publishing relational data as XML documents. The first is the need for a *language* to specify the conversion from relational data to XML documents. The second is the need for an *implementation* to efficiently carry out the conversion. The language specification describes how to structure and tag data from one or more tables as a hierarchical XML document. One of this paper's contributions is a language specification based on SQL, with minor extensions in the form of new scalar and aggregate functions for XML document construction. These extensions can be easily added to existing relational systems without departing from existing SQL semantics. Also, as a result of extending SQL in this manner, standard APIs like ODBC can be used to query and retrieve XML documents. This allows existing tools and applications to easily integrate relational data and XML documents.

Given a language specification for converting relational tables to XML documents, an implementation to carry out the conversion raises many challenges. Relational tables are flat, while XML documents are tagged, hierarchical and graph-structured. What is the best way to go from the former to the latter? In order to answer this question, we characterize the space of alternatives based on whether tagging and structuring are done early or late in query processing. We then refine this space based on how much processing is done inside the relational engine and explore various alternatives within this space. Our performance comparison of the alternatives using a commercial database system (DB2) shows that an "unsorted outer union" approach – based on late tagging and late structuring – is attractive when the resulting XML document fits in main memory, while a "sorted outer union" approach – based on late tagging and early structuring – performs well otherwise. Our results also show that constructing an XML document inside a relational engine is far more efficient than doing so outside the engine. Thus, constructing an XML document inside the relational engine has a two-fold advantage – not only does it allow existing SQL APIs to be reused for XML documents, but it is also much more efficient.

The rest of this paper is organized as follows. In Section 2 we provide a brief overview of XML and in Section 3 we present our SQL-based language approach for publishing relational data as XML. In Section 4 we explore a range of implementation alternatives and in Section 5 we evaluate the performance of the alternatives and show the superiority of the "outer union" plans. In Section 6 we outline the algorithm to generate the "outer union" plans from the SQL query specification proposed in this paper. In Section 7 we discuss related work, and in Section 8, we present our conclusions and ideas for future work.

```
<customer id="C1">
   <name> John Doe </name>
   <accounts>
       <account id="A1"> 1894654 </account>
       <account id="A2"> 3849342 </account>
   </accounts>
   <porders>
       <porder id="P01" acct="A2">    // first purchase order
           <date> 1 January 2000 </date>
           <items>
               <item id="I1"> Shoes </item>
               <item id="I2"> Bungee Ropes </item>
           </items>
           <payments>
               <payment id="P1"> due January 15 </payment>
               <payment id="P2"> due January 20 </payment>
               <payment id="P3"> due February 15 </payment>
           </payments>
   </porders>
   <porder id="P02" acct="A1">    // second purchase order
       ....
   </porder>
</customer>
```

Figure 1: An XML Document Describing a Customer

2. An XML Primer

Extensible Markup Language (XML) [26] is a hierarchical format for information exchange in the World Wide Web. An XML document consists of nested element structures starting with the root element. Each element has a tag associated with it. In addition to nested elements, an element can have attributes and values or sub-elements. Figure 1 shows an XML document representing a customer in a simple e-commerce application, where each customer has a set of accounts and a set of purchase orders, and each purchase order in turn has a set of items and a set of payments. The customer is represented by the <customer> element, which appears at the root of the document. The customer has an id attribute, which is a special kind of attribute that uniquely identifies an element in an XML document. Each customer has a name, represented by the <name> sub-element nested under customer. A customer element also has nested sub-elements representing the accounts and purchase orders associated with the customer. Each of these has other attributes and sub-elements.

An interesting feature to note in Figure 1 is that the purchase order elements have an attribute called "acct". This is a field that is of type IDREF (such typing information is specified in a Document Type Descriptor [26] – not shown here – associated with an XML document), and it logically points to an element having the same value as its ID. Thus, the first purchase order points to the second account, while the second purchase order points to the first account. Another key feature of the XML model is that elements can be ordered. For example, purchase orders could be ordered by date to make the most recent purchases appear first in the document. More details on XML can be found in [26].

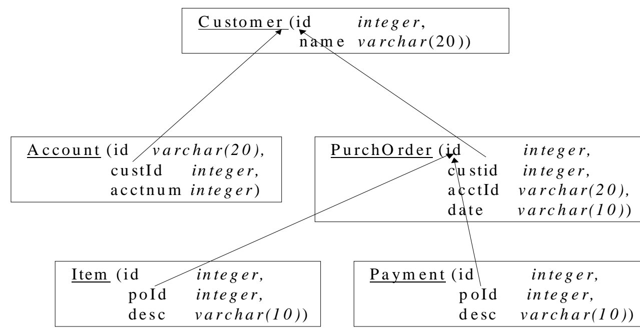

Figure 2: Customer Relational Schema

3. A SQL-based Language Specification

A key requirement for converting relational data to XML documents is a language to specify the conversion. Our approach to designing this langauge is to harness and extend the power of SQL for this purpose. Nested SQL statements are used to specify nesting, and SQL functions are used to specify XML element construction.

Consider the relational schema shown in Figure **2**, which models the customer information of Figure 1 in relational form. As shown, there are customer, account, purchase order, item and payment tables. Each table has an id and other attributes associated with it, and there are foreign key relationships (shown by means of arrows) relating the tables. To convert data in this relational schema to the XML document in Figure 1, we can write a SQL query that follows the nested structure of the document, as shown in Figure 3.

The query in Figure 3 produces both SQL and XML data – each result tuple contains a customer's name together with the XML representation of the customer. The overall query consists of several correlated sub-queries. The easiest way to understand the query is to look at it from the top down. The top-level query retrieves each customer from the customer table. For each customer, a correlated sub-query is used to retrieve the customer's accounts (lines 2-4) and purchase orders (lines 5-14). Assume for the moment that each correlated sub-query returns an XML document fragment. The next step then is to create the customer XML elements. This is done by calling the CUST XML constructor (lines 1-14), which takes a customer name, account information (in XML form), and purchase order information (in XML form) as input and produces a customer XML element as output.

The definition of the CUST XML constructor is shown in Figure 4. Conceptually, it should be viewed as a scalar function returning XML. For each input tuple, CUST tags the columns as specified and produces an XML fragment.

The correlated sub-queries can be interpreted similarly, with the ACCT, PORDER, ITEM and PAYMENT constructors defined much like CUST. Each nested query finally has to

```
01. Select  cust.name, CUST(cust.id, cust.name,
02.                            (Select  XMLAGG(ACCT(acct.id, acct.acctnum))
03.                             From    Account acct
04.                             Where acct.custId = cust.id),
05.                            (Select  XMLAGG(PORDER(porder.id, porder.acct, porder.date,
06.                                                   (Select  XMLAGG(ITEM(item.id, item.desc))
07.                                                    From    Item item
08.                                                    Where item.poId = porder.id),
09.                                                   (Select  XMLAGG(PAYMENT(pay.id, pay.desc))
10.                                                    From    Payment pay
11.                                                    Where pay.poId = porder.id)))
12.                                  group order by porder.date
13.                             From    PurchOrder porder
14.                             Where  porder.custId = cust.id))
15. From  Customer cust
```

Figure 3: SQL Query to Construct XML Documents from Relational Data

```
create function CUST (custId: integer, custName: varchar(20), acctList: xml, porderList: xml)
   returns xml language xml return
        <customer id={custId}>
              <name> {custName} </name>
              <accounts> {acctList} </accounts>
              <porders> {porderList} </porders>
        </customer>
```

Figure 4: Definition of an XML Constructor

return one XML fragment. This is done using the aggregate function XMLAGG, which concatenates the XML fragments (e.g., ITEM fragments) produced by XML constructors.

To order XML fragments, the XMLAGG aggregate function needs to work on ordered inputs. For example, to order all the purchase orders associated with a customer by their date, we need to ensure that the XMLAGG aggregate function in line 5 of Figure 3 aggregates the purchase orders in that order. Since ordered inputs to aggregate functions are not currently supported in SQL, we propose an extension to SQL that would make this possible. Using our extension, a "group order by" clause is used in conjunction with an order-sensitive aggregate function to specify the order in which the aggregate function is to operate on its inputs. This use of the "group order by" clause is illustrated in line 12 of Figure 3, where the purchase orders of a customer are ordered by their date before being aggregated by the XMLAGG function.

4. Implementation Alternatives

In the previous section we presented one possible language for specifying the conversion from relational data to XML documents. The rest of the paper is more general in scope – we examine different *implementations* to carry out the conversion, independently of the specification language.

In order to understand the various alternatives for publishing relational data as XML documents, we characterize the solution space based on the main differences between

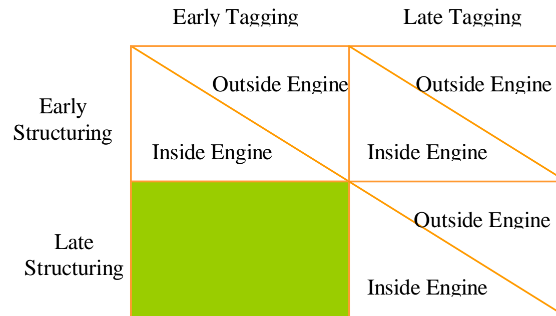

Figure 5: Space of Alternatives for Publishing XML

relational tables and XML documents, namely, XML documents have *tags* and *nested structure,* while relational tables do not. Thus, in converting from relational tables to XML documents, tags and structure have to be added somewhere along the way. One approach is to do tagging as the final step of query processing (*late tagging*), while another approach is to do it earlier in the process (*early tagging*). Similarly, structuring can be done as the final step of query processing (*late structuring*) or it can be done earlier (*early structuring*). These two dimensions of tagging and structuring give rise to a space of alternatives shown pictorially in Figure 5.

Each alternative in this space has variants depending on how much work is done inside the relational engine. Note that "inside the engine" means that tagging and structuring are done *completely inside* the relational engine, whereas "outside the engine" means that part, though not necessarily all, of that work is done outside the relational engine. Also note that early tagging with late structuring is not a viable alternative because adding tags to an XML document without having its structure makes no sense. We now explore the space of alternatives in detail by means of concrete examples.

4.1. Early Tagging, Early Structuring

In this class of alternatives, tagging and structuring are both done early in query processing. We first describe an "outside the engine" approach, where a significant amount of processing is done as a stored procedure, and then we describe two approaches where more processing is done inside the relational engine.

4.1.1. The Stored Procedure Approach

Perhaps the simplest technique for structuring relational data as an XML document is for an application or stored procedure to iteratively issue a nested set of queries that matches the structure of the desired XML document. Consider the XML document example shown in Figure 1. First a query is issued to retrieve the desired root level elements (customers).

Information about a customer such as their customer ID and customer name are retrieved and tagged. Then, using the customer's ID, a query is issued to retrieve the customer's account information, which is then tagged and output. Next, while still on the same customer, a query is issued to retrieve the customer's purchase orders ordered by their date. This ensures that the purchase orders associated with the customer are in the desired order. For each purchase order retrieved, a separate query is then issued for the purchase order's items and the purchase order's payment information. Once this is done, the processing for one customer is complete. The same procedure is repeated for the next customer until the entire XML document has been constructed.

The Stored Procedure approach essentially performs a nested-loop join outside the engine by issuing queries for each nested structure within the desired XML document. It falls under the category of early structuring because the queries that are issued mimic the structure of the result. Also, since tagging is done as soon as each nested structure becomes available, this approach falls under the category of early tagging.

Although the Stored Procedure approach is commonly used today, a major problem with it is that one or more SQL queries are issued *per tuple* for tables that have nested structures in the resulting XML document. Thus, to construct large documents, *thousands* of queries may need to be issued. The overhead of issuing so many queries can cause serious inefficiencies, as will be confirmed by the performance study in Section 5. Another significant problem with this approach is that it always dictates both a particular join order and the nested-loop join method even when other join orders and/or join methods might be superior.

4.1.2. The Correlated CLOB Approach

One way to eliminate the overhead of issuing many SQL queries is to move processing inside the relational engine so that one large query with sub-queries, rather than many top-level queries, is executed. The challenge is then to have the relational engine tag and build up the nested structures so that processing which was previously performed in a stored procedure now occurs inside the engine. This can be accomplished by adding engine support for the XML constructors and XMLAGG function that we described in Section 3. The query to produce the XML result can then be executed as a nested SQL query. The query's execution would basically follow the language specification shown in Figure 3 by executing correlated sub-queries for nested queries. This is depicted pictorially in Figure 6. Since the XML document fragments created by the XML constructors (such as CUST() and ACCT()) can be of arbitrary size, the obvious choice is to represent them as large objects, such as Character Large Objects (CLOBs), inside the relational engine.

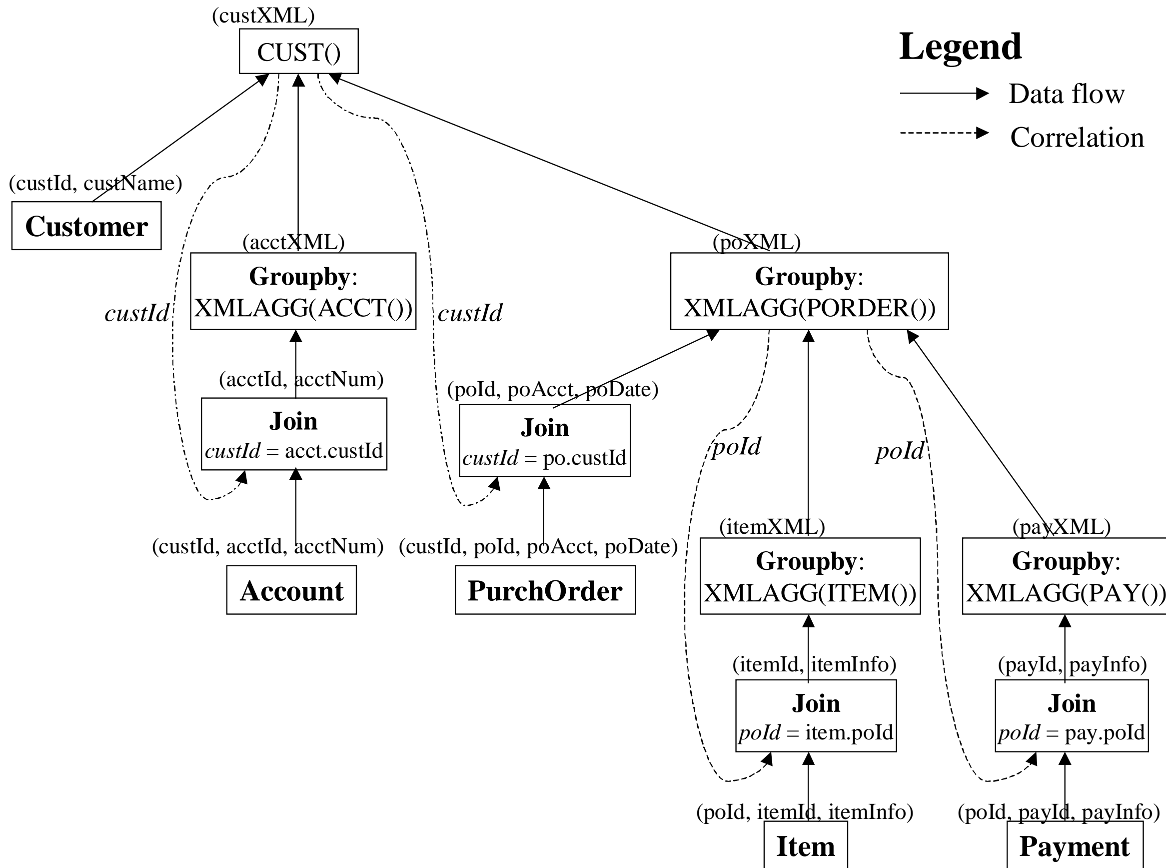

Figure 6: SQL Query Execution Plan for the Correlated CLOB Approach

Because of correlation during execution, the Correlated CLOB approach still performs a nested-loop join. However, it is likely to out-perform the Stored Procedure approach because a single query is issued to the relational engine. Nonetheless, the fact that intermediate XML structures are represented as CLOBs can lead to performance problems. This is because CLOB columns are typically stored separately from the tuples they belong to. Thus, in parallel environments, fetching CLOBs (scattered around different nodes) can lead to significant performance degradation. Further, CLOBs often need to be written to a separate storage area on disk during sorts. Finally, each invocation of an XML constructor copies its inputs, which may include CLOBs, to a new CLOB. This repeated creation and copying of CLOBs can be costly.

4.1.3. The Decorrelated CLOB Approach

One disadvantage of the Correlated CLOB approach is that, because of its correlated sub-queries, it naturally implies a nested-loop join strategy. This can be avoided by performing query de-correlation [21] inside the relational engine to give the relational optimizer more flexibility. The query execution plan obtained by de-correlating the query in Figure 3 is shown in Figure 7.

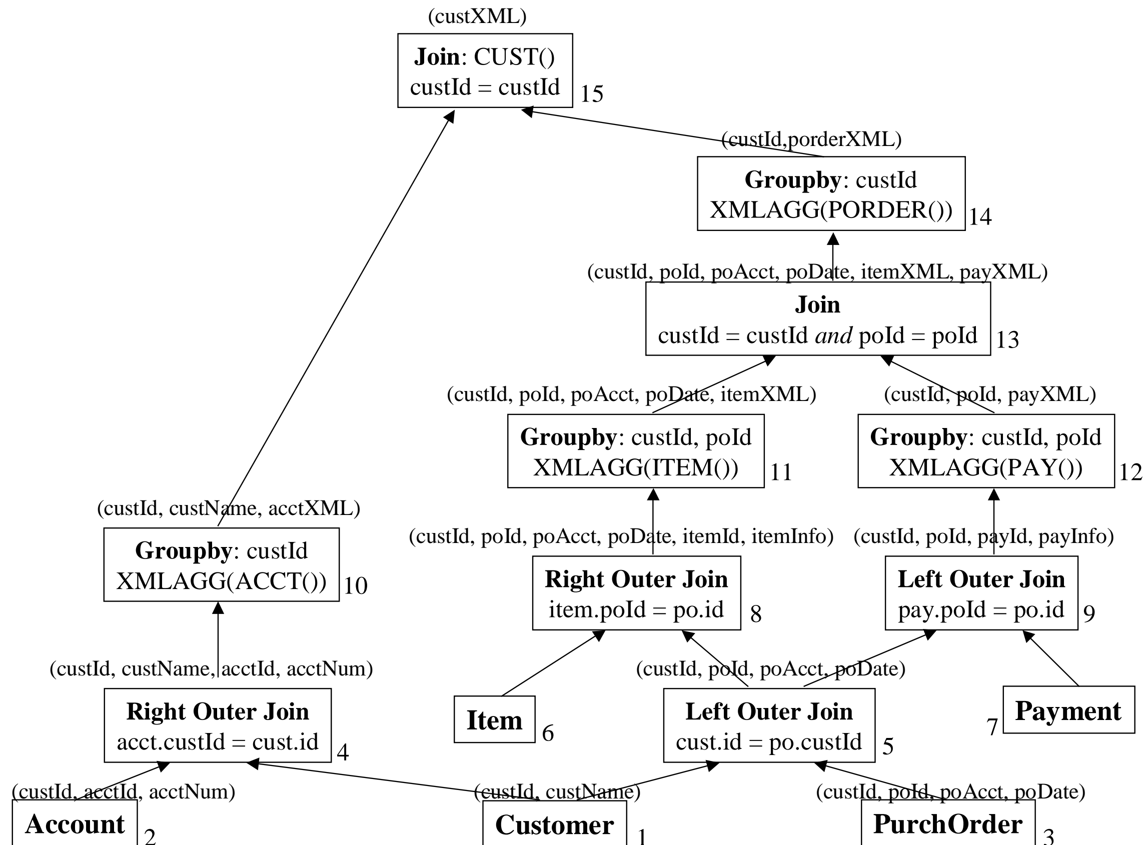

Figure 7: SQL Query Execution Plan for the De-correlated CLOB Approach

To create the de-correlated query, first each path from the root-level table to a leaf-level table is computed by joining the tables along the path. In our example, these join paths are (a) Customer joined with Account, (b) Customer joined with Purchase Order joined with Item and (c) Customer joined with Purchase Order joined with Payment. These join paths are represented by boxes 4, 8 and 9 respectively in Figure 7. Outer joins are used because the information about a parent has to be preserved even if it has no children (for example, an XML element for a customer should be produced even if there are no accounts associated with that customer). Where possible, common sub-expressions are used so that redundant computation is avoided. Thus, for example, the join between Customers and Purchase Orders is shared between two path computations.

Once the root to leaf paths are computed, the set of leaf-level XML elements corresponding to each leaf-level table is then built up. This is done by tagging the leaf-level XML elements and then aggregating them by grouping on the id columns (e.g., custId and poId) of the ancestor tables on the path from the root-level table to the leaf-level table. This is done in boxes 10, 11 and 12 in Figure 7. Higher-level structures are built up by joining on these id fields and using an XML constructor. This is done till the root level is reached. (boxes 13-15 in Figure 7).

```
Select cust.id, cust.name, acct.id, acct.num, po.id, po.acctId, po.date, item.id, item.info, pay.id, pay.info
From  Customer cust
        left join Account acct on cust.id = acct.custId
        left join PurchOrder po on cust.id = po.custId
        left join Item item on po.id = item.poId
        left join Payment pay on po.id = pay.poId
```

Figure 8: SQL Query for the Redundant Relation Approach

Despite the fact that this approach is more flexible in allowing the engine to explore join strategies, it shares the same problems as the Correlated CLOB approach with respect to repeated copying, parallelism and materialization of CLOBs. This is because tagging and structuring are done early, thus creating large, opaque intermediate objects. Is it possible to defer tagging and structuring to arrive at a more efficient alternative? We explore this class of alternatives next.

4.2. Late Tagging, Late Structuring

In the class of alternatives that defer tagging and structuring, both tagging and structuring are done as the final step of constructing an XML document. The construction of an XML document is therefore logically split into two phases: (a) content creation, where relational data is produced, and (b) tagging and structuring, where the relational data is structured and tagged to produce the XML document. We first deal with content creation. We consider only "inside the engine" approaches so that database functionality, such as joins, can be exploited.

4.2.1. Content Creation: The Redundant Relation Approach

One simple way to produce the needed content is to join all of the source tables using join predicates to relate parents to their children. In our example, this would be done by joining the Customer, Accounts, Purchase Order, Item and Payment tables using the relevant predicates. The SQL query to do this is shown in Figure 8.

This approach has the advantage of using regular, set-oriented relational processing, but it also has a serious pitfall – it has both content and processing redundancy. To see this, consider what the result of the query in Figure 8 would look like. Each customer's account information would be repeated PO × IT × PA times, where PO is the number of purchase orders associated with the customer, IT is the number of items per purchase order, and PA is the number of payments per purchase order. The problem here is that multi-valued data dependencies [9] are created when we try to represent a hierarchical structure as a single table. This increases both the size of the result and the amount of processing to produce it, both of which are likely to severely impact performance.

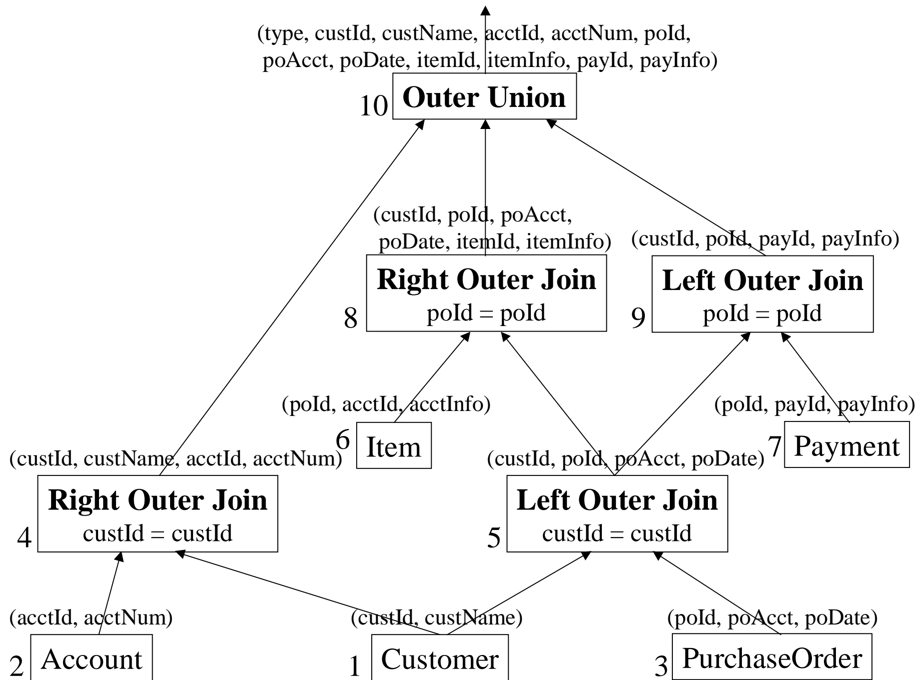

Figure 9: SQL Query Execution Plan for the (Unsorted) Path Outer Union Approach

4.2.2. Content Creation: The (Unsorted) Path Outer Union Approach

The basic problem with the Redundant Relation approach is that the number of tuples in the relational result grows as the *product* of the number of children per parent. If we could limit the result's size to be the *sum* of the number of children per parent, redundancy would be reduced dramatically. To do this, we need to separate the representation of a given child of a parent from the representation of the other children of the same parent. For example, one tuple of the relational result should represent *either* an account *or* a purchase order associated with the customer, not both.

Figure 10 shows a SQL query that reduces content redundancy for the query of Figure 3. The query execution plan corresponding to this SQL query is also shown in Figure 9. First, as in the De-Correlated CLOB approach, each path from the root-level table to a leaf-level table is computed by means of joins. In our example query, there are three such paths – Customer-Account, Customer-PurchaseOrder-Item and Customer-PurchaseOrder-Payment. Thus, Customers are joined with Accounts (one path), Customers are joined with Purchase Orders which are in turn joined with Items (another path) and Customers are joined with Purchase Orders which are in turn joined with Payments (final path). These join paths are represented by boxes 4, 8 and 9 respectively in Figure 9, and correspond to lines 5-8, 13-17 and 18-21 respectively in the SQL query of Figure 10 (these are defined as the inline views custAcct, custPorderItem and custPorderPay in the SQL query using the "with" statement [1][3]). As in

```
-- First compute all the paths from the root to the leaves
01. with cust (custId integer, custName varchar(20)) as (
02.    select cust.id, cust.name
03.    from Customer cust
04. ),
05. custAcct (custId integer, custName varchar(20), acctId integer, acctNum integer) as (
06.    select cust.id, cust.name, acct.id, acct.acctnum
07.    from Account acct right join cust on (acct.custId = cust.id)
08. ),
09. custPorder (custId integer, poId integer, poAcct varchar(20), poDate varchar(10)) as (
10.    select cust.id, po.id, po.acctId, po.date
11.    from cust left join Purchorder po on (cust.id = po.custId)
12. ),
13. custPorderItem (custId integer, poId integer, poAcct varchar(20), poDate varchar(10), itemId integer,
14.                         itemInfo varchar(20)) as (
15.    select custpo.custId, custpo.poId, custpo.poAcct, custpo.poDate, item.id, item.info
16.    from Item item right join custPorder custpo on (item.poId = custpo.poId)
17. ),
18. custPorderPay (custId integer, poId integer, payId integer, payInfo varchar(20)) as (
19.    select custpo.custId, custpo.poId, pay.id, pay.info
20.    from custPorder custpo left join Payment pay on (custpo.poId = pay.poId)
21. ),
22. -- The following is the main query which performs the (path) outer union
23. select 0, custId, custName, acctId, acctName, null, null, null, null, null, null, null
24. from custAcct
25.    union all
26. select 1, custId, null, null, null, poId, poAcct, poDate, itemId, itemInfo, null, null
27. from custPorderItem
28.    union all
29. select 2, custId, null, null, null, poId, null, null, null, null, payId, payInfo
30. from custPorderPay
```

Figure 10: SQL Query for the (Unsorted) Path Outer Union Approach

the De-correlated CLOB approach, common sub-expressions are used so that redundant computation is avoided where possible. Thus, the join between Customers and Purchase Orders is shared between two path computations.

Each join path produces one tuple per data item in the leaf level of the XML tree. Each tuple describing a leaf level data item also includes information about its ancestors in the XML tree (see the lists of columns above each join box in Figure 9). A separate tuple describing a parent needs to be present only if the parent has no children. The use of outer joins to relate a parent with its children ensures this semantics.

The final step in the process of creating the relational content is to glue together all the tuples representing leaf level elements in the XML tree into a single relation. The obvious way to do this is to union the content corresponding to each leaf level element. There are, however, some complications with this strategy since the tuples corresponding to different leaf level elements need not have the same number or types of columns. For example, tuples representing accounts have only four columns, while tuples representing items have six columns. In order to handle this heterogeneity, a separate column is allocated in the union's output for each distinct column in the union's input. For each tuple representing a particular

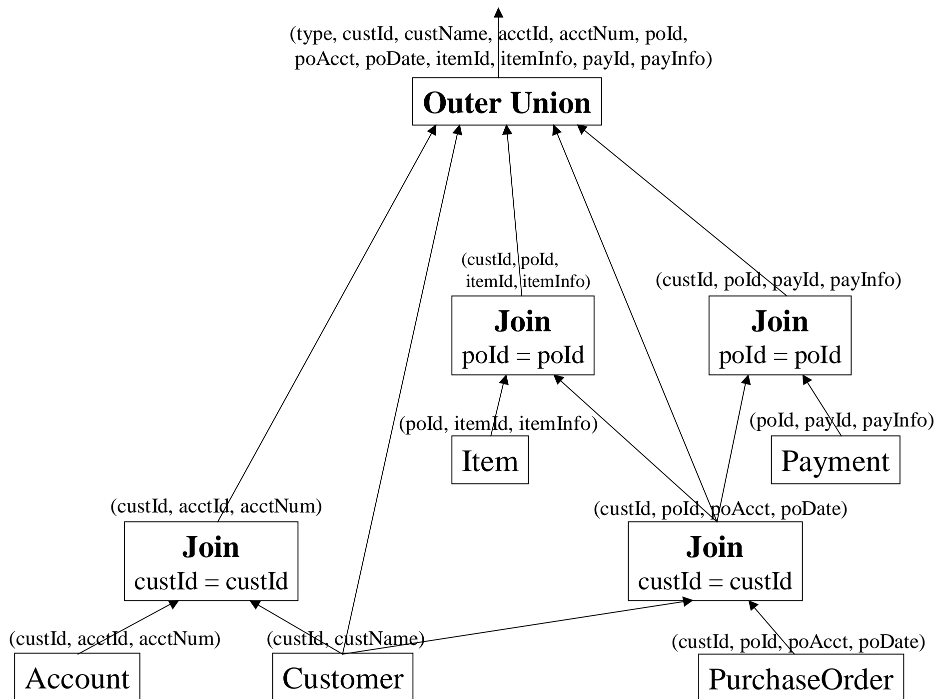

Figure 11: SQL Query Execution Plan for the (Unsorted) Node Outer Union Approach

leaf level element and its ancestors, only a subset of these columns will be used and the rest will be set to null (hence the name *outer union* by analogy to outer join). This is done by the outer union box in Figure 9 and corresponds to lines 23-30 in Figure 10.

To keep track of the origin of each tuple, e.g. to distinguish an account tuple from an item tuple, a type column is added to the result of the outer union as well. We call this approach the *Path Outer Union* approach because it computes each *path* from the root-level table to a leaf-level table and *outer unions* them.

4.2.3. Content Creation: The (Unsorted) Node Outer Union Approach

The Path Outer Union approach that was just described eliminates much of the data and computational redundancy of the Redundant Relation approach. This is because children of the same parent are represented in separate tuples. However, there is still some data redundancy present. In particular, all parent information is replicated with every child tuple. For example, a customer's information, such as full name, address, etc., is replicated in every account associated with the customer. One way to get around this is to feed the parent information directly into the outer union operator and to carry only the parent ids along with the children (and their descendants). We refer to this option as the *Node Outer Union* approach to distinguish it from the preceding Path Outer Union approach.

Figure 11 shows the query execution plan for the Node Outer Union approach for our running example. Figure 12 shows the corresponding SQL query. Note how all the parent

```
-- First compute all the paths from the root to the leaves
01. with cust (custId integer, custName varchar(20)) as (
02.    select cust.id, cust.name
03.    from Customer cust
04. ),
05. custAcct (custId integer, acctId integer, acctNum integer) as (
02.    select cust.id, acct.id, acct.acctnum
03.    from Account acct, cust
04.    where acct.custId = cust.id
05. ),
06. custPorder (custId integer, poId integer, poAcct varchar(20), poDate varchar(10)) as (
07.    select cust.id, po.id, po.acctId, po.date
08.    from cust, Purchorder po
09.    where cust.id = po.custId
10. ),
11. custPorderItem (custId integer, poId integer, itemId integer, itemInfo varchar(20)) as (
12.    select custpo.custId, custpo.poId, item.id, item.info
13.    from Item item, custPorder custpo
14.    where item.poId = custpo.poId
15. ),
16. custPorderPay (custId integer, poId integer, payId integer, payInfo varchar(20)) as (
17.    select custpo.custId, custpo.poId, pay.id, pay.info
18.    from custPorder custpo, Payment pay
19.    where custpo.poId = pay.poId
20. ),
21. -- The following is the main query which performs the (node) outer union
22. select 0, custId, custName, null, null, null, null, null, null, null, null, null
23. from custRoot
24     union all
25. select 1, custId, null, acctId, acctName, null, null, null, null, null, null, null
26. from custAcct
27.    union all
28. select 2, custId, null, null, null, poId, poAcct, poDate, null, null, null, null
29. from custPorder
30     union all
31. select 3, custId, null, null, null, poId, null, null, itemId, itemInfo, null, null
32. from custPorderItem
33.    union all
34. select 4, custId, null, null, null, poId, null, null, null, null, payId, payInfo
35. from custPorderPay
```

Figure 12: SQL Query for the (Unsorted) Node Outer Union Approach

information (customer, and customer-purchase order information) is fed directly to the outer union operator and how only the ids of the parents (customer id and purchase order id) are propagated along with their respective children and descendants. Since all the parent information is fed directly to the outer union operator, it is sufficient to perform a regular join (as opposed to an outer join) to relate a parent to its children without potentially losing information.

The Node Outer Union approach reduces data redundancy as compared to the Path Outer Union approach because information about a parent is not replicated with all the children. However, the Node Outer Union approach increases the number of tuples in the result because each parent is now represented by a separate tuple. One concern with both of the Outer Union approaches is that the number of columns in the result increases with the depth

and width of the XML document. Though only a subset of the columns in a given tuple will have data values, in the absence of null value compression, this may lead to increased processing overhead due to larger tuple widths.

4.2.4. Structuring/Tagging: The Hash-based Tagger

In the previous three sections, we discussed techniques to produce the relational content necessary for creating an XML document. The final step in the Late Structuring Late Tagging alternatives is to tag and structure the relational content to form the results. This can be done either inside or outside the relational engine. If it is performed inside the relational engine, it can be implemented as an aggregate function. Such a function would be invoked as the last processing step, after the relational content has been produced. This (single) aggregate function would logically perform the function of all the XML constructors and XMLAGGs in the user query. This would ensure that large objects are not carried around during processing, which is one of the potential disadvantages of the CLOB approaches.

In order to tag and structure the results, either inside or outside the engine, we need to do two things: (a) group all siblings in the desired XML document under the same parent (and eliminate duplicates in the case of the Redundant Relation approach) and (b) extract information from each tuple and tag it to produce the XML result. An efficient way to group siblings is to use a main-memory hash table to look up the parent of a node, given the parent's type and id information (including the ids of ancestors of the parent).

Whenever a tuple containing information about an XML element is seen, it is hashed on the element's type and the ids of its ancestors in order to determine whether its parent is already present in the hash table. If the parent is present, a new XML element is created and added as a child of the parent. If the parent is not present (note that this is possible because the result tuples do no appear in any particular order), then a hash is performed on the type and ids of all ancestors *except* that of the parent. This is to determine if the grandparent exists. If the grandparent is present, a place-holder is created for the parent (to be filled in with the parent tuple when it arrives) and then the child is created under the place-holder for the parent. If the grandparent is also not present, the procedure is repeated until an ancestor is present in the hash table or the root of the document is reached.

After all the input tuples have been hashed, the entire tagged structured result can be written out as an XML file. If a specific order is required for the elements of the resulting XML document, such as ordering purchase orders by their date, then that order can either be maintained as children are added to a parent or it can be enforced by a final sort before writing out the XML document.

The main limitation of using a hash-based tagger is that performance can degrade rapidly when there is insufficient memory to hold the hash table and the intermediate result. However, it may be possible to partition the data into memory-sized chunks, much like in a hash join [23]. Exactly how to do this partitioning (and merging) is left for future work.

4.3. Late Tagging, Early Structuring

The main problem with the Late Tagging Late Structuring approaches we just considered is that complex memory management needs to be performed by the hash-based tagger when memory is scarce. To eliminate this problem, the relational engine can be used to produce "structured relational content", which can then be tagged in *constant space*. We first explore a technique to produce structured content before describing the constant space tagger.

4.3.1. Structured Content Creation: The Sorted Outer Union Approaches

The key to structuring the relational content is to order it the way that it needs to appear in the result XML document. This can be achieved by ensuring that:

1) *All of the information about a node X in the XML tree occurs either before or along with the information about the children of X in the XML tree.* This essentially says that parent information occurs before, or with, child information.
2) *All tuples representing information about a node X and its descendants in an XML tree occur contiguously in the tuple stream.* This ensures that information about a particular node and its descendants is not mixed in with information about non-descendant nodes.
3) *All tuples representing information about a node X of a given type in the XML tree occur before any tuples representing information about a sibling node of X of a different type that appears after X in the XML tree.* This ensures that siblings of different types will appear in the desired order because order is significant in an XML document. For example, all accounts associated with a customer should occur before all purchase orders associated with the same customer.
4) *The relative order of the tuples matches that of any user-specified order.* This rule is included to handle user-defined ordering requests.

We now show that performing a single (final) sort of the unstructured relational content is sufficient to ensure these properties. Our discussion here will be based on the (Unsorted) Node Outer Union approach for constructing unstructured relational content. The solution for the Path Outer Union Approach is actually simpler because it always satisfies (the along-with case of) condition 1. It will also be easy to see how the technique generalizes to the Redundant Relation approach.

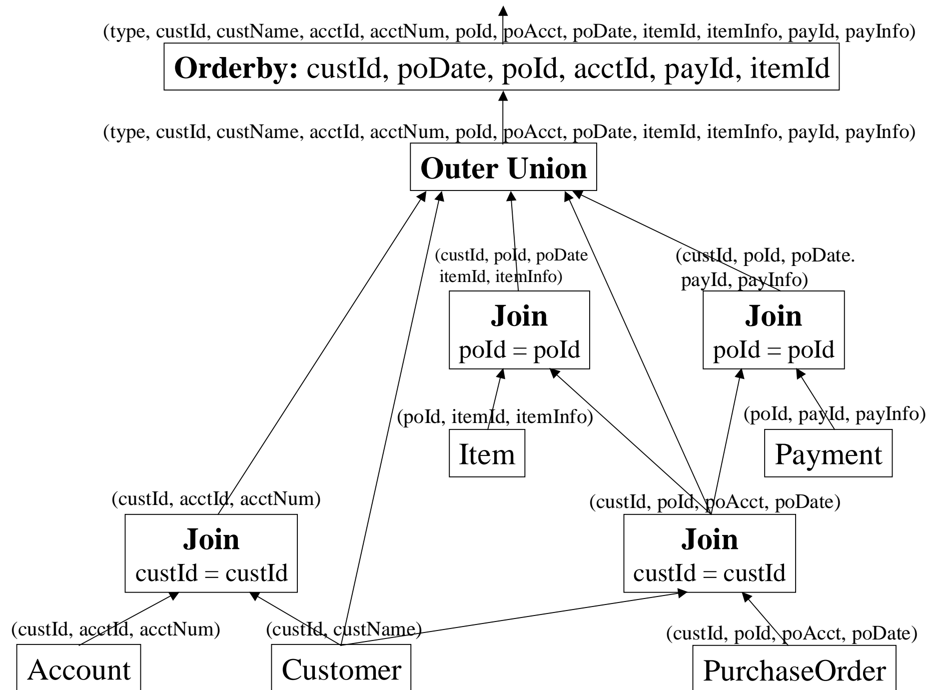

Figure 13: SQL Query Execution Plan for the Sorted Node Outer Union Approach

To ensure that conditions 1 through 4 above are satisfied, all that is required is to sort the result of the Node Outer Union on the id fields and the user-specified sort fields such that (a) the id field of a parent node occurs before the id fields of its children nodes in the sort sequence, (b) the id fields of sibling nodes appear in the sort sequence in the reverse order as the siblings are to appear in the XML document (the significance of the reverse sort sequence will be explained shortly), and (c) the user defined order fields on a node (if any) appear immediately before the id field of that node in the sort sequence. Thus, in our running example, sorting the node outer union result on the sort sequence (CustId, PODate, POId, AcctId, PaymentId, ItemId) will ensure that result is in document order. The query execution plan performing this sort and the corresponding SQL query are shown in Figure 13 and Figure 14 respectively.

For correctness, it is important to propagate the user-specified sort fields (purchase order date) of a parent (purchase order) to all its descendants (items and payments) before performing the outer union, as shown in Figure 13. It is also important that tuples having null values in the sort fields occur before tuples having non-null values (i.e., nulls must sort low). As we shall explain next, this is necessary to ensure that parents and siblings appear in the desired order.

Let us now see how the above mentioned sort order ensures that conditions 1 through 4 are satisfied. Condition 1 will be satisfied because a tuple corresponding to a parent node (say, customer) will have null values for the child id columns (say, account id). Since we

```
-- Lines 1-37 compute the node outer union as in Figure 12
01. with cust (custId integer, custName varchar(20)) as (
02.    select cust.id, cust.name
03.    from Customer cust
04. ),
05. custAcct (custId integer, acctId integer, acctNum integer) as (
02.    select cust.id, acct.id, acct.acctnum
03.    from Account acct, cust
04.    where acct.custId = cust.id
05. ),
06. custPorder (custId integer, poId integer, poAcct varchar(20), poDate varchar(10)) as (
07.    select cust.id, po.id, po.acctId, po.date
08.    from cust, custRoot po
09.    where cust.id = po.custId
10. ),
11. custPorderItem (custId integer, poId integer, poDate varchar(10), itemId integer, itemInfo varchar(20)) as (
12.    select custpo.custId, custpo.poId, custpo.poDate, item.id, item.info
13.    from Item item, custPorder custpo
14.    where item.poId = custpo.poId
15. ),
16. custPorderPay (custId integer, poId integer, poDate varchar(10), payId integer, payInfo varchar(20)) as (
17.    select custpo.custId, custpo.poId, custpo.poDate, pay.id, pay.info
18.    from custPorder custpo, Payment pay
19.    where custpo.poId = pay.poId
20. ),
21. outerUnion (type integer, custId integer, custName varchar(20), acctId integer, acctNum integer,
22.                 poId integer, poAcct varchar(20), poDate varchar(10), itemId integer, itemInfo varchar(20),
23.                 payId integer, payInfo varchar(20)) as
23.      select 0, custId, custName, null, null, null, null, null, null, null, null, null
24.      from cust
25.         union all
26.      select 1, custId, null, acctId, acctName, null, null, null, null, null, null, null
27.      from custAcct
28.         union all
29.      select 2, custId, null, null, null, poId, poAcct, poDate, null, null, null, null
30.      from custPorder
31.         union all
32.      select 3, custId, null, null, null, poId, null, poDate, itemId, itemInfo, null, null
33.      from custPorderItem
34.         union all
35.      select 4, custId, null, null, null, poId, null, poDate, null, null, payId, payInfo
36.      from custPorderPay
37. ),
38. -- This is the main query that sorts the outer union result
39. select type, custId, custName, acctId, acctNum, poId, poAcct, poDate, itemId, itemInfo, payId, payInfo
40. from outerUnion
41. order by custId, poDate, poId, acctId, payId, itemId
```

Figure 14: SQL Query for the Sorted Node Outer Union Approach

ensure that tuples with null values in their sort columns occur first, parent tuples (customers) will always occur before child tuples (accounts). Also, condition 2 is satisfied because the parent's id (customer id) occurs before a child's id (account id) in the sort sequence, thus ensuring that the children of a parent node are grouped together after the parent.

Condition 3 is satisifed because the ids of the siblings appear in the reverse order in the sort sequence as the siblings are to appear in the result XML document, and because nulls sort low. To see why this is the case, let us return to our example, where a customer's accounts

need to occur before the customer's purchase orders in the result XML document. By ensuring that the purchase order id occurs before account id in the sort sequence, and that nulls sort low, tuples representing accounts (which are tuples with null values for purchase order id) occur before tuples representing purchase orders (which are tuples having non-null values for purchase order id). Finally, condition 4 is satsified because user-defined sort fields (purchase order date) are added immediately before the id (purchase order id) of the node being ordered in the sort sequence. It is important to propagate the sort fields (purchase order date) of a parent (purchase order) to all its descendants (items and payments) before performing the outer union because this ensures that all the descendants of the parent are sorted in the same way as the parent and thus prevents condition 2 from being violated.

The Sorted Outer Union approaches have the advantage of scaling to large data volumes because relational database sorting algorithms are designed to be "disk-friendly". These approaches can also produce user-specified orderings with little additional cost. However, they do do more work than necessary; a total order is always produced even when only a partial order is needed. This is because we do not require elements of the same type (say, accounts) to be ordered in the absence of user-specified ordering requirements.

4.3.2. Tagging Sorted Data: The Constant Space Tagger

Once the structured relational content has been created, as described in the previous two sections, the final step is to tag and construct the result XML document. Since tuples arrive in document order, they can be immediately tagged and written out as they are seen. The tagger only requires enough memory to remember the parent ids of the last tuple seen. These ids are used to detect when all the children of a particular parent node have been seen so that the closing tag associated with the parent can be written out. For example, after all the items and payments of a purchase order have been seen, the closing tag for purchase order (</porder>) has to be written out. To detect this, the tagger stores the id of the current purchase order and compares it with that of the next tuple. It should be clear that the storage required by the constant space tagger is proportional only to the level of nesting and is independent of the size of the XML document.

5. Performance Comparison of Alternatives for Publishing XML

We have outlined a number of alternatives for creating XML documents from a relational database. These are summarized in Figure 15. Our qualitative assessments indicate that every alternative has some potential disadvantage. In this section, we will conduct a performance evaluation of the alternatives to determine which ones are likely to win in practice (and in

Classification		Approach	Short Name	Description	Potential Problems
Early Tag *Early* Structure	Outside Engine	Stored Procedure	Stored Proc	Issues separate queries according to document structure, essentially doing nested loops joins outside the engine.	1) Many SQL queries. 2) Fixed join strategy (nested loops join).
	Inside Engine	Correlated CLOB	CLOB-Corr	The "inside the engine" equivalent of the Stored Procedure approach. Uses CLOBs to build up intermediate XML fragments.	1) Fixed join strategy (nested loops join). 2) Intermediate CLOBs created during query processing.
	Inside Engine	De-Correlated CLOB	CLOB-DeCorr	De-correlated version of CLOB-Corr. Also requires CLOBs for intermediate fragments.	1) Intermediate CLOBs created during query processing.
Late Tag *Late* Structure	Inside or Outside Engine	Redundant Relation	Redundant R (In/Out)	Creates a relation with data redundancy because each child of a parent is repeated many times.	1) Data redundancy. 2) Memory overflow in hash-based tagger.
	Inside or Outside Engine	Unsorted Path Outer Union	Unsorted OU (In/Out)	Creates "outer union" of leaf elements and avoids data redundancy. Inside and outside engine versions of hash-based structuring/tagging.	1) Data redundancy (on a smaller scale). 2) Memory overflow in hash-based tagger. 3) Wide tuples
	Inside or Outside Engine	Unsorted Node Outer Union	Unsorted NOU (In/Out)	Similar to Unsorted OU, but also includes tuples for non-leaf elements in outer union.	1) Memory overflow in hash-based tagger. 2) Wide tuples
Late Tag *Early* Structure	Inside or Outside Engine	Sorted Path Outer Union	Sorted OU (In/Out)	Structures the results of Unsorted OU by sorting it in document order.	1) Data redundancy (on a smaller scale). 2) Wide tuples. 3) Requires total order of relational result.
	Inside or Outside Engine	Sorted Node Outer Union	Sorted NOU (In/Out)	Structures the results of Unsorted NOU by sorting it in document order.	1) Wide tuples. 2) Requires total order of relational result.

Figure 15: Summary of Approaches for Publishing XML

what situations). Towards this end, we will first identify a set of parameters that are simple and yet can model a wide range of relational to XML conversions. In the experiments reported below, we do not consider queries with user-defined sort orders.

5.1. Modeling Relational to XML Transformations

In order to study the performance effects of converting flat relational data to nested XML documents, we will vary the nature of nesting of the queries that specify the construction of XML documents (see Figure 3 for an example query). In our experiments, the nesting of queries is characterized by two parameters. The first parameter is the *query fan out.* This corresponds to the maximum number of sub-queries directly nested under a parent (sub) query. For example, the query in Figure 3 has a query fan out of two because the (sub) queries in lines 1-15 and lines 5-14 each have two directly nested sub-queries (lines 2-4, 5-14 and lines 6-8, 9-12, respectively) while the other sub-queries (lines 2-4, 6-8, 9-12) have no directly nested sub-queries. The second parameter used to characterize nesting is *query depth.* This corresponds to the maximum nesting level of sub-queries. In our example in Figure 3,

the query depth is three because there are three levels of query nesting – the first being the top level query (lines 1-15), the second being the queries in lines 2-4 and 5-14 and the third being the queries in lines 6-8 and 9-12.

In our experiments, we only consider "balanced" queries, where (a) each non-leaf (sub) query has the same number of directly nested sub-queries and (b) all leaf (sub) queries are at the same depth. This results in a simple set of parameters, each of which can be studied in isolation. Note that the query in Figure 3 is not balanced because it satisfies condition 1 but not condition 2. It is important to note that the query fan out and query depth do not directly specify the fan out or the depth of the result XML document. Even at low values of query fan out and query depth, the result XML document can be wide/deep depending on the XML constructors used (see Figure 4). The query fan out and query depth only specify the structure of the repeating "set" sub-elements, such as the accounts associated with a customer.

Our goal here is to study the effects of nesting relational data as XML documents, and not the complexity of the SQL used to create the data for an XML element. Hence, for this performance study, the relational schema we use for the experiments will mirror the nesting of the SQL query specifying the construction (e.g., like Figure 3 and Figure 2) and each relation in the schema will be a base table. Thus, the same parameters (query fan out and query depth) used to vary the structure of the query are also used to vary the structure of the underlying relational schema. Each table has an ID field, which is its primary key. It also has a PID (parent id) field that serves as a foreign key for its parent. To match parents with their children, a join is specified between the ID and PID field of the parent and child tables, respectively. In addition to these two fields, each table has two data fields. The first is an integer field (IntVal) while the second is a 20 character long string field (CharVal).

We now identify two additional parameters that, given a schema, suffice to describe a specific experimental database instance. The first parameter is the *number of roots*, which specifies the number of tuples present in the table at the schema tree's root level. The second parameter is the *number of leaf tuples*, which specifies the total number of tuples present in all of the leaf-level tables combined. The number of tuples in each leaf-level table is thus the number of leaf tuples divided by the number of leaf-level tables. These two parameters together determine another important derivative parameter, the *instance fan out*, which specifies the number of children tuples of each type that a parent tuple has under the assumption that every parent tuple has the same number of child tuples of a given type.

We have chosen to use the number of leaf tuples as the primary parameter and the instance fan out as a derivative parameter because the overall number of leaf tuples (where

Parameter	**Description**
Query Fan Out	Number of sub-queries directly nested under a parent sub-query. This is also a measures the "bushiness" of the underlying relational schema and the result XML document.
Query Depth	Number of levels of nesting of sub-queries. This is also a measure of the "depth" of the underlying relational schema and the result XML document.
Number of Roots	Number of tuples in the root level table in the relational schema. This is also a measure of the number of root-level XML elements.
Number of Leaf Tuples	Number of tuples in all the leaf tables in the relational schema combined. This is also a measure of the size of the result XML document.

Figure 16: Experimental Parameters

the bulk of the data resides) is directly related to the size of the XML document produced. Thus, holding the number of leaf tuples constant allows us to study how the different approaches behave when (essentially) the same amount of data is structured differently. The experimental parameters for our performance study are summarized in Figure 16.

We now characterize the XML document result created for a given experimental relational database instance. The integer and character column values of each tuple in the relational database instance are tagged as XML elements each having a tag name that is 3 characters long. The XML fragments of child tuples are nested under the XML representation of the parent tuples. The result is always a single XML document. This was done to make the experimental results easy to interpret. Note that we do not explicitly consider selections on tables since the same performance effects can be explored by varying the number of roots and the number of leaf tuples.

5.2. Experimental Setup

To conduct our performance comparison, we implemented the various alternatives discussed in Section 4 in the code base of the DB2 Universal Database system [3]. The XML constructors and XMLAGG were implemented as new built-in functions. The Stored Procedure approach was implemented as an "unfenced" stored procedure, i.e., it ran in the same address space as the relational database engine, to maximize performance. The other "outside the engine" approaches were each implemented as local embedded-SQL programs, running on the same machine as the database server, to avoid unpredictable network delays. We implemented the "outside the engine" approaches as stored procedures as well, but since this did not significantly change their performance, those results are not included here. A driver program, implemented as a local embedded-SQL program, was used to time the results on a warm DB2 cache. The XML result was always written out as an NT file. All experiments were performed on a 366 MHz Pentium II processor with 256 MB of main memory running Windows NT 4.0.

Parameter	**Range of Values**	**Default**
Query Fan Out	2, 3, 4	2
Query Depth	2, 3, 4	2
# Roots	1, 50, 500, 5000, 40000	5000
# Leaf Tuples	160000, 320000, 480000	320000

Figure 17: Parameter Settings for Experiments

For the experiments, we varied the parameters discussed in Section 5.1 in the ways shown in Figure 17. For each experiment, we varied one of these parameters and used the default values for the rest. This enabled us to determine the effect of each parameter on performance. Indexes were created on the ID and PID fields for all of the tables in the relational schema. Detailed optimizer statistics were collected for each table and index before any queries were run. For most experiments, the sort heap and buffer pool sizes were set so that all processing was done in main memory (the maximum data/XML document size was 25MB); the one exception is the experiments in Section 5.7, where the effect of reduced memory is considered. Since the Node and Path Outer Union approaches behave similarly in a wide range of situations, we only show the performance for the Path Outer Union for most of the studies. The relative performance of the Node and Path Outer Union approaches is discussed separately in Section 5.8.

5.3. Inside the Engine vs. Outside the Engine Approaches

To get an initial feel for the results, we first explore the effects of varying the query fan out while holding the other parameters constant. The resulting time taken to construct the XML document for the "inside the engine" and the "outside the engine" approaches is shown in Figure 18 and Figure 19, respectively. The Redundant Relation approach is not shown in these graphs because it performs very poorly with increasing fan out due to large data redundancy. In fact, the time for *just executing* the associated relational query, ignoring the time for tagging and writing the XML result to disk, was about 155 seconds at a query fan out of 4. The performance of the Redundant Relation approach was among the worst of all possible approaches throughout all of our experiments, so we will not examine it further.

The interesting thing to note in Figure 18 and Figure 19 is that while the Stored Procedure approach incurs a significant overhead because it issues many queries to the relational engine, the Correlated CLOB approach, its "inside the engine" counterpart, takes roughly one-third of the time. This actually points to a more general trend. For the Unsorted and Sorted Outer Union approaches as well, the "inside the engine" versions take less than half the time to execute than their corresponding "outside the engine" versions. In order to explain these results, we need to break down the time for creating XML document results.

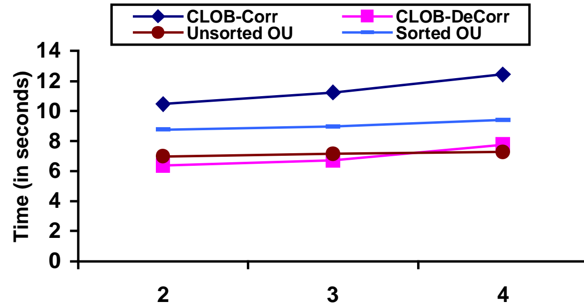

Figure 18: Varying Query Fan Out (Inside the Engine)

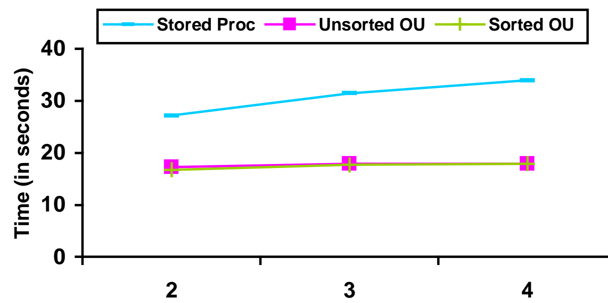

Figure 19: Varying Query Fan Out (Outside the Engine)

For the "outside the engine" approaches, there are four components to generating the XML result: 1) the time to produce the relational content, either structured or unstructured, 2) the time to bind out the rows of the relational content into host variables outside the engine, 3) the time to tag and possibly structure the relational result, and 4) the time to write the XML result out to a file. For the "inside the engine" approaches, there are the same components except that virtually no time is spent binding out the results. We measured each of these components independently for the various approaches. The tagging time for the CLOB approaches was not separated because it forms an integral part of the content computation.

Figure 20 shows this time break down for each approach and it is easy to see that the time to bind out (copy) tuples to host variables from the relational engine dominates the cost of the "outside the engine" approaches. These results were found hold regardless of whether the bind out was done in a local client program or within an unfenced stored procedure. Moreover, increasing the size of the communication buffer between the client application and the database server so that larger portions of the result could be copied over to the client address space in one chunk did not significantly reduce the bind-out cost. On the other hand, the "inside the engine" approaches eliminate the host variable bind-out cost for every tuple; their only bind-out is done for the final (single) result document. Consequently, the "inside the engine" approaches perform much better. This points to our first firm conclusion –XML document construction should be done inside the engine to maximize performance.

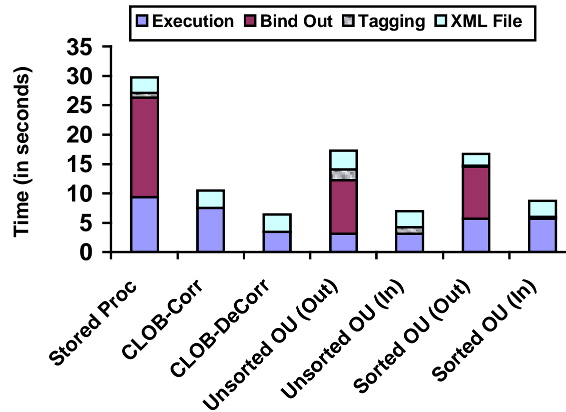

Figure 20: Break Down of XML Construction Time

Since the "inside the engine" approaches consistently outperform the "outside the engine" approaches, the rest of our experimental results will consider these approaches separately. Note that despite their poor relative performance, the "outside the engine" approaches are valuable to consider because they can be used with relational database systems that do not have support for the new XML scalar/aggregate functions proposed in this paper.

5.4. Effect of Query Fan Out

We now return to Figure 18 and Figure 19 for the purpose of examining the effect of varying the query fan out. For the "inside the engine" techniques, increasing the query fan out increases the time for producing the XML result, as shown in Figure 18. This is not surprising since increasing the query fan out increases the number of joins that need to be performed. What is more interesting is the relative performance of the different approaches. The Correlated CLOB approach, which utilizes many correlated sub-queries, performs worse than the other set-oriented plans. This is because the relational optimizer has no choice but to use the nested loop join strategy. Among the Outer Union based plans, the Unsorted Outer Union approach is more efficient than the Sorted Outer Union approach. This implies that the cost of sorting (and using a simple constant space tagger) is more expensive than avoiding the sort and using a more complex hash-based tagger (given sufficient main memory).

A rather surprising result is that the De-Correlated CLOB approach, despite having to repeatedly copy information and carry CLOBs during computation, performs fairly well and in fact, is the best strategy for low query fan outs. This is because the DB2 optimizer picked a query plan whereby CLOBs could be retained in main memory without having to be materialized. Also, since the query depth is low, the overhead of repeatedly copying CLOBs is not significant.

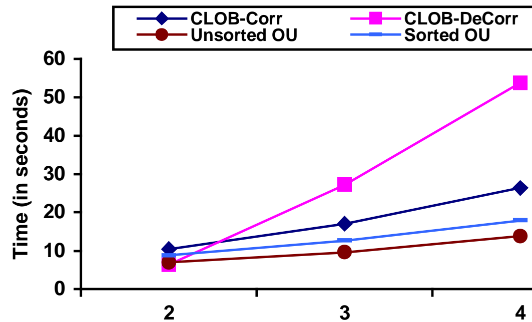

Figure 21: Varying Query Depth (Inside the Engine)

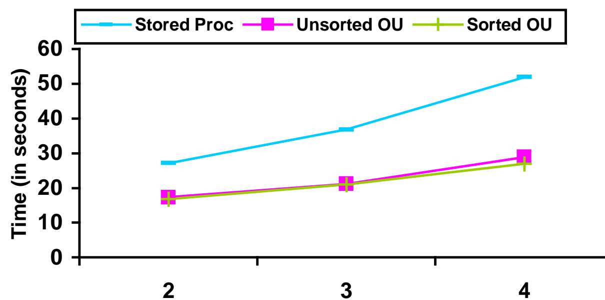

Figure 22: Varying Query Depth (Outside the Engine)

Figure 19 shows the effects of query fan out on the "outside the engine" approaches. The Stored Procedure approach performs much worse than the Outer Union approaches because of the overhead of issuing many separate database queries and using a fixed join strategy. Surprisingly, unlike for the "inside the engine" case, the execution times for the Sorted and Unsorted Outer Union approaches are approximately the same here (even though the Sorted Outer Union Approach has the extra overhead of the sort). This is because the constant space tagger is a streaming operator; i.e., it produces a part of the XML document as soon as it sees a tuple. It can thus overlap tagging with writing the XML document to disk, whereas the hash-based tagger has to process all input tuples before writing anything to disk.

5.5. Effect of Query Depth

We now turn our attention to the next parameter – query depth. Figure 21 shows the effect of varying the query depth parameter for the "inside the engine" approaches. While the execution time for all the approaches increases with query depth, it is interesting to note the dramatic increase for the De-Correlated CLOB approach. This is because, not too surprisingly, the relational query optimizer makes mistakes when dealing with very complex queries at higher values of query depth. For instance, the query for a producing an XML document of query depth 4 has 15 aggregations (XMLAGGs) and 12 joins! In these cases, the optimizer makes some poor decisions such as choosing to sort after an aggregation. This requires CLOBs to be written to a temporary space and materialized again later. This problem

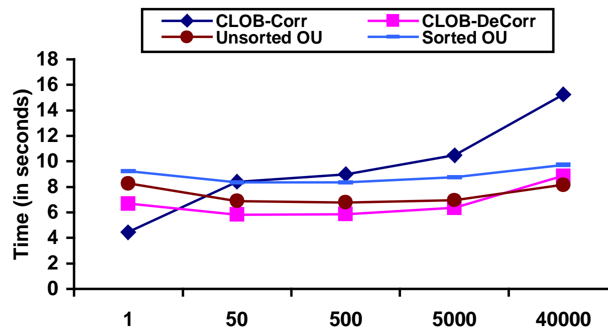

Figure 23: Varying Number of Roots (Inside the Engine)

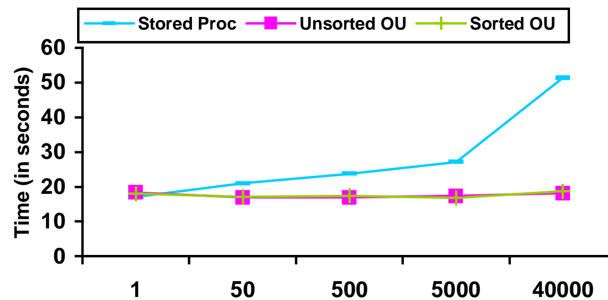

Figure 24: Varying Number of Roots (Outside the Engine)

is compounded by the fact that the XMLAGG aggregate function is opaque to DB2's traditional relational database optimizer, so it has no good way to estimate and consider the size of the CLOB result.

The effects of varying query depth for the "outside the engine" approaches are similar to those for the corresponding "inside the engine" approaches. This is shown in Figure 22.

5.6. Effect of Number of Roots

The next parameter of interest is the number of roots. At low values for this parameter, the performance of the Correlated CLOB approach improves dramatically, relative to the other "inside the engine" approaches (see Figure 23). This happens because only two correlated sub-queries have to be issued for constructing the XML document with one root element.

A similar effect occurs (for similar reasons) with the Stored Procedure approach, the "outside the engine" counterpart of the Correlated CLOB approach (see Figure 24). The relative performance of the outer union approaches remains unchanged.

5.7. Effect of Number of Leaf Tuples vs. Memory Size

For the next set of experiments, we varied the overall size of the data set by varying the number of leaf tuples. When there was sufficient memory, the relative performance of the various approaches did not change. However, when the amount of memory available for processing was reduced so that the XML document construction could not be performed entirely in main memory, (a) the performance of the CLOB approaches degraded even further

because of disk-resident CLOBs, and (b) the Unsorted Outer Union approaches were unable to proceed because our hash-based tagger cannot (currently) handle overflows. In contrast, the Sorted Outer Union approaches, which are based on the highly scalable relational sort, adapted gracefully.

5.8. Path Outer Unions vs. Node Outer Unions

We now compare the performance of the Node and Path Outer Union approaches. As mentioned earlier, their performance is nearly identical when there is sufficient main memory. In fact, despite its data redundancy, the Path Outer Union approach performs slightly better (by less than a second) because there are fewer tuples to process (and thus to bind out in case of the "outside the engine" approaches).

The main difference between the two outer union approaches occurs when memory is scarce. In this case, for bushy trees (having high instance fan out) the Node Outer Union approaches perform better – a difference of up to three seconds – while for non-bushy trees (having low instance fan out), the Path Outer Union approaches perform better. This is because there is greater data redundancy in the Path Outer Union approach for bushy trees, and the overhead of its having to spill the extra data to disk exceeds the advantage of processing fewer tuples.

5.9. Summary of Experimental Results

To summarize, our performance comparison of the alternatives for publishing XML documents points to the following conclusions:

1) Constructing XML documents inside the relational engine is far more efficient that doing so outside. This is mainly because of the high cost of binding out tuples to host variables.
2) When processing can be done in main memory, the Unsorted Outer Union approach is stable and always among the very best (both inside and outside the engine).
3) When processing cannot be done in main memory, the Sorted Outer Union approach is the approach of choice (both inside and outside the engine). This is because the relational sort operator scales well.

It is worth noting at this point that the potential disadvantage of outer union approaches – that of "wide" tuples (see Figure 15) – does not significantly impact their performance. The main reasons for this is that, for a given outer union result tuple, most of the column values are null. Efficient null compression is thus able to reduce the overhead of carrying around many columns during query processing. (For a discussion of performance on database systems that do not do efficient null compression, see Section 7 under the heading "Middleware queries".)

6. Algorithms to Generate SQL Queries for the Outer Union Approaches

The results of our performance evaluation show that the outer union approaches proposed in this paper provide a stable and efficient way to retrieve and structure the relational data needed to construct an XML document. In this section, we present algorithms that can be used to automatically generate outer union SQL queries. In particular, we present translation algorithms that take as input a SQL query specifying the construction of XML documents (using the SQL language extensions proposed in this paper) and produce as output an outer union SQL query that generates the content for the result XML documents.

The algorithms presented here are applicable both inside the relational engine as well as in an application layer outside the relational engine. If the relational engine provides support for the SQL language extensions proposed in this paper, the SQL query rewrite module [16] can use the algorithms to generate outer union plans for efficient query execution. Otherwise, an application program can parse the user query and use the algorithms to generate outer union SQL queries. Since the outer union SQL queries do not use any XML-specific extensions, they can be executed by a standard relational database engine.

While our focus will be on describing the algorithms in the context of the SQL query extensions proposed in this paper, it should be easy to see how the algorithms generalize to other query languages that use nested sub-queries to create nested document structures. Examples of such query languages include XQuery [28], XML-QL [7] and RXL [11].

The rest of this section is organized as follows. We first describe the parameters that are passed as input to the outer union SQL generation algorithms, before describing the algorithms themselves.

6.1. Input Parameters for Outer Union SQL Generation

Using the language extensions proposed in this paper, a SQL query specifying the construction of XML documents can be represented as the template shown in Figure 25. This template SQL query has the following six parameters: *sqlCols* is the list of SQL columns produced as the result of the query, *xmlConstructor* is the name of the XML constructor used to produce the XML output, *xmlCols* is the list of SQL columns used in *xmlConstructor* to produce the XML output, *subQueries* is the list of SQL sub-queries used to build up intermediate XML fragments needed for producing the XML output, *fromList* is the list of tables referred to in the from clause of the SQL query, and *predicates* is the list of predicates present in the where clause of the SQL query.

```
select sqlCols, xmlConstructor(xmlCols, subQueries)
from fromList
where predicates
```

Figure 25: Template of Top-level SQL Query Specifying XML Construction

```
select XMLAGG(xmlConstructor(xmlCols, subQueries))
        group order by orderingCols
from fromList
where predicates
```

Figure 26: Template of SQL Sub-Query Specifying XML Construction

As an example, consider the SQL query shown in Figure 3. For this query, the six parameters are: *sqlCols* = [cust.id], *xmlConstructor* = CUST, *xmlCols* = [cust.id, cust.name], *subQueries* = [<acct sub-query>, <porder sub-query>], *fromList* = [Customer as cust], *predicates* = []. Note that *predicates* is represented as the empty list because the example SQL query does not have a where clause.

Each sub-query in the *subQueries* list conforms to the template shown in Figure 26. Here the parameters *xmlConstructor*, *xmlCols*, *subQueries*, *fromList*, and *predicates* have the same semantics as before. *orderingCols* is the list of SQL columns used to specify the order in which XML elements are to be aggregated.

As an example, the SQL sub-query in Figure 3 that constructs purchase order XML elements (lines 5-14) has the following parameters: *xmlConstructor* = PORDER, *xmlCols* = [porder.id, porder.acct, porder.date], *subQueries* = [], *orderingCols* = [porder.date], *fromList* = [PurchOrder as porder], *predicates* = [porder.custId = cust.id]).

Using the input parameters described above, we now outline algorithms to generate outer union SQL queries from SQL queries specifying the construction of XML documents. We first outline the algorithms for generating path outer union SQL queries before turning our attention to algorithms for generating node outer union SQL queries.

6.2. Generating SQL Queries for the Path Outer Union Approaches

As described in Sections 4.2.2 and 4.3.1, the basic idea in the path outer union approaches is to compute all paths from the root level tables to the leaf level tables by means of joins. The results of these join paths are then outer unioned together to produce the desired relational content. In the case of the sorted path outer union approach, the outer unioned results are also sorted on the key and ordering columns.

```
01. Algorithm buildPaths (SQLQuery sqlQuery, String parentName, Boolean firstChild) returns QueryString
02.   // Check whether sqlQuery is a top-level query or a sub-query
03.   if  ( sqlQuery is a top-level query )  then
04.       // Start with creating the root of the paths
05.       resultString = "with " + sqlQuery.name + " (" + <output columns and types> + ") as ("
06.       resultString += "select " + sqlQuery.sqlCols + ", " + sqlQuery.xmlCols
07.       resultString += "from " + sqlQuery.fromList
08.       resultString += "where " + sqlQuery.predicates
09.       resultString += ")"
10.   else
11.       // sqlQuery is a sub-query. Create an intermediate result that is outer joined with the parent. Propagate
12.       // parent information if this is the first child
13.       resultString = ", " + sqlQuery.name + " (" + <output columns and types> + ") as ("
14.
15.      // Check whether this is the first child. If so, propagate parent's data columns
16.       if (firstChild) then
17.           resultString += "select " + <parent's ids and all data columns> + ", " + sqlQuery.xmlCols
18.       else
19.           resultString += "select " + <parent's ids and ordering columns> + ", " + sqlQuery.xmlCols
20.       endif
21.
22.       // Perform outer join with parent
23.       resultString += "from " + parentName + "left join " + sqlQuery.fromList
24.       resultString += "on (" + sqlQuery.predicates + ")"
25.       resultString += ")"
26.   endif
27.
28.   // Recurse on all sub-queries of sqlQuery to produce paths till the leaf level
29.   for (each subQuery in sqlQuery.subQueries) do
30.        if (subQuery is first child query) then
31.            resultString += buildPaths(subQuery, sqlQuery.name, true) // Propagate parent's data columns
32.        else
33.            resultString += buildPaths(subQuery, sqlQuery.name, false)
34.        endif
35.   endfor
36.
37.   // Return the result string
38.   return resultString
```

Figure 27: Algorithm to Generate Paths for the Path Outer Union Approaches

Figure 27 shows the algorithm to generate the SQL for computing the paths from the root level tables to the leaf level tables. As shown, the algorithm takes the user-defined SQL query as input. The algorithm also takes in two other input parameters, which are used during recursive invocations of the algorithm.

We will now walk through the algorithm using the example query shown in Figure 3 and illustrate how the SQL paths (lines 1-21) are generated for the path outer union query shown in Figure 10. The algorithm buildPath is first invoked with the *sqlQuery* parameter set to be equal to the top-level SQL query in Figure 3. The other two parameters, *parentName* and *firstChild,* are set to be equal to the values null and false, respectively. Since *sqlQuery* is a top-level query, the "if" branch of the conditional is executed (lines 4-9 in Figure 27) and this generates the cust inline view in the SQL query of Figure 10 (lines 1-4). Note that we use the

notation *sqlQuery.x* to refer to the parameter *x* in the template representation of *sqlQuery*. Also, for ease of exposition, we have assumed the presence of an extra field in *sqlQuery*, name, that has the name of the inline view being generated (here name has the value “cust”).

Once the inline view for the top-level query is created, the algorithm is invoked recursively on all the sub-queries (lines 29-35 of Figure 27). In our example, the sub-queries are those that produce the account and purchase order XML fragments corresponding to a customer. Since in the path outer union approach, all the parent’s data columns have to be propagated with one of its children, a boolean flag (*firstChild*) is set in the recursive call invocation. This indicates which child is to propagate the parent data columns. The name of the parent inline view (cust) is also passed as a parameter (*parentName*) so that the children can join with the parent on the path from the root to the leaf.

On a recursive invocation of the algorithm on a sub-query (such as for the account and purchase order sub-queries), the else branch of the first condition is executed (lines 11-25 in Figure 27). This produces the SQL that propagates parent data columns if necessary (lines 16-20) and creates an outer join to relate the parent and the child (lines 22-25). In our example, the recursive invocations produce the custAcct and custPorder inline views (lines 5-12 in Figure 10). Further recursive invocations of the algorithm produce the other inline views, namely custPorderItem and custPorderPay.

Once the SQL for the paths from the root level tables to the leaf level tables are generated, the next step in generating SQL for the unsorted path outer union approach is to outer union these paths (lines 23-30 in Figure 10). The high-level pseudo-code for generating the SQL for the outer union is given in Figure 28. This algorithm is invoked with the top-level SQL query. First, all the leaf sub-queries are determined. In our example, these are the sub-queries corresponding to accounts, items and payments. Then, for each leaf-level sub-query, a separate leg of the outer union is created (lines 5-18 in Figure 28). Each leg of the outer union has the appropriate type information to identify the leg (lines 11-12) and draws results from the appropriate root-to-leaf path (lines 16-17).

The complete algorithm to generate the SQL for the unsorted path outer union approach is given in Figure 29. As can be seen, it first invokes the buildPaths function to build all root to leaf paths, and then invokes the buildPathsOuterUnion function to outer union the results.

The algorithm for generating the SQL for the sorted path outer union approach is not presented here because it is actually a simplified version of the corresponding algorithm for the sorted node outer union approach. This will be discussed in the next section in the context of SQL generation for the node outer union approaches.

```
01. Algorithm buildPathOuterUnion (SQLQuery sqlQuery) returns QueryString
02.    // Outer union all the root to leaf paths
03.    leafSubQueries = Get all leaf sub-queries of sqlQuery
04.    numLeafSubQueries = size(leafSubQueries)
05.    for (index = 0; index < numLeafSubQueries; ++index) do
06.        // Create one leg of the outer union
07.        if (index > 0) then
08.            resultString += " union all "
09.        endif
10.
11.        // Add the type field for the path
12.        resultString += "select " + index + ", "
13.
14.       // Add the other fields and create the from clause
15.       currSubQuery = leafSubQueries[index]
16.       resultString += <all columns of currSubQuery with null padding where necessary>
17.       resultString += "from " + currSubQuery.name
18.    endfor
19.
20.    // Return the result string
21.    return resultString
```

Figure 28: Algorithm to Generate the Outer Union for the Path Outer Union Approaches

```
01. Algorithm buildUnsortedPathOuterUnionSQL (SQLQuery sqlQuery) returns QueryString
02.    // First build the paths from the root-level tables to the leaf-level tables
03.    resultString = buildPaths(sqlQuery, null, false)
04.
05.    // Next, outer union the paths
06.    resultString += buildPathOuterUnion(sqlQuery)
07.
08.    // Return the SQL string constructed
09.    return resultString
```

Figure 29: Algorithm to Generate SQL for the Unsorted Path Outer Union Approach

6.3. Generating SQL Queries for the Node Outer Union Approaches

As described in Sections 4.2.3 and 4.3.1, the main difference between the path and node outer union approaches is that the latter avoids some data redundancy by feeding parent information directly to the outer union. Thus only the parent id and ordering columns have to be carried along with the children. This difference between the path and node outer union approaches results in different SQL queries for the two approaches (for example, see Figure 10 and Figure 12) and hence requires different SQL generation algorithms. In this section, we thus present algorithms for generating SQL queries for the node outer union approaches.

As in the path outer union approaches, the first step in generating SQL queries for the node outer union approaches is to generate paths from the root level tables to the leaf level tables (lines 1-20 in Figure 12 and Figure 14). The algorithm to generate the desired paths if given in Figure 30. This algorithm is broadly similar to the corresponding algorithm for the path outer union approaches (see Figure 27). There are, however, two important differences. First, the algorithm for the node outer union approaches does not have the logic to propagate

```
01. Algorithm buildPaths (SQLQuery sqlQuery, String parentName) returns QueryString
02.   // Check whether sqlQuery is a top-level query or a sub-query
03.   if (sqlQuery is a top-level query) then
04.      // Start with creating the root of the paths
05.      resultString = "with " + sqlQuery.name + " (" + <output columns and types> + ") as ("
06.      resultString += "select " + sqlQuery.sqlCols + ", " + sqlQuery.xmlCols
07.      resultString += "from " + sqlQuery.fromList
08.      resultString += "where " + sqlQuery.predicates
09.      resultString += ")"
10.   else
11.      // sqlQuery is a sub-query. Create an intermediate result that is outer joined with the parent. Propagate
12.      // parent information if this is the first child
13.      resultString = ", " + sqlQuery.name + " (" + <output columns and types> + ") as ("
14.
15.     // Propagate output columns
16.     resultString += "select " + <parent's ids and ordering columns> + ", " + sqlQuery.xmlCols
17.
18.      // Join with parent
19.      resultString += "from " + parentName + ", " + sqlQuery.fromList
20.      resultString += "where " + sqlQuery.predicates
21.      resultString += ")"
22.   endif
23.
24.   // Recurse on all sub-queries of sqlQuery to produce paths till the leaf level
25.   for (each subQuery in sqlQuery.subQueries) do
26.      resultString += buildPaths(subQuery, sqlQuery.name)
27.   endfor
28.
29.   // Return the result string
30.   return resultString
```

Figure 30: Algorithm to Generate Paths for the Node Outer Union Approaches

parent data values along with children. This is because only the parent ids and ordering columns need to be propagated for node outer union approaches (line 16). Second, the node outer union approaches employ regular joins to relate parents and children, as opposed to the outer joins used in the path outer union approaches. This is shown in lines 19-21 of Figure 30.

The next step in SQL generation for the unsorted node outer union approach is to outer union the paths generated in the previous step (to generate lines 21-35 in Figure 12). This algorithm (not shown) is very similar to the corresponding algorithm for the unsorted path outer union approach (Figure 28), but with one key difference. Instead of creating an outer union consisting of only root to leaf paths, the algorithm creates an outer union of all paths from the root (including paths to intermediate nodes). This is a direct consequence of having to feed parent information directly to the node outer union. The complete algorithm to generate the SQL for the unsorted node outer union approach is shown in Figure 31.

We now turn our attention to generating SQL for the sorted node outer union approach. As described in Section 4.3.1, the sorted node outer union approach essentially sorts the results of the unsorted node outer union approach on the appropriate columns so that the results appear in document order. Figure 32 shows the algorithm for generating the SQL

```
01. Algorithm buildUnsortedNodeOuterUnionSQL (SQLQuery sqlQuery) returns QueryString
02.    // First build the paths from the root-level tables
03.    resultString = buildPaths(sqlQuery, null)
04.
05.    // Next, outer union the paths
06.    resultString += buildNodeOuterUnion(sqlQuery)
07.
08.    // Return the SQL string constructed
09.    return resultString
```

Figure 31: Algorithm to Generate SQL for the Unsorted Node Outer Union Approach

```
01. Algorithm buildSortedNodeOuterUnionSQL (SQLQuery sqlQuery) returns QueryString
02.    // First build the paths from the root level tables
03.    resultString = buildPaths(sqlQuery, null)
04.
05.    // Create the outer union as an inline view
06.    resultString += ", outerUnion (" + <output columns and types> + ") as ("
07.    resultString += buildNodeOuterUnion(sqlQuery)
08.    resultString += ")"
09.
10.    // Now create the main query the orders the outer union result
11.    resultString += "select " + <output column names>
12.    resultString += "from outerUnion"
13.    resultString += "order by "
14.
15.    // Create the correct sort sequence
16.    for (each subQuery in breadth first traversal of all sub-queries of sqlQuery,
17.                                  traversing siblings in right to left order) do
18.         resultString += subQuery.orderingCols + ", " + subQuery.ids
19.    endfor
20.
21.    // Return the SQL string constructed
22.    return resultString
```

Figure 32: Algorithm to Generate SQL for the Sorted Node Outer Union Approach

query for the sorted node outer union approach. The first few steps (lines 2-8) essentially produce the SQL for the unsorted outer union approach, except that the result of the outer union is created as an inline view. This part of the algorithm produces lines 1-37 of the sorted outer union SQL query in Figure 14.

The next part of the SQL generation algorithm (lines 10-22 in Figure 32) creates the main SQL query that sorts the unsorted outer union result in the desired order. This produces lines 38-41 of the SQL query in Figure 14. In order to create the right ordering sequence, which satisfies the conditions outlined in Section 4.3.1, all the sub-queries are traversed in a breadth first manner, starting with the top-level query. This ensures that parent ids appear before child ids in the sort order. Further, all the siblings are traversed in right to left order so that the id columns of siblings appears in the sort sequence in the reverse order as the siblings appear in the result XML documents. Finally, the ordering columns associated with a sub-query appear before the ids of the sub-query. This ensures that user-specified ordering requirements are satisfied in the final output.

The SQL generation algorithm for the sorted path outer union approach is very similar to the corresponding algorithm for the sorted node outer union approach described above. The only difference between the two is that in the sorted path outer union approach, the SQL for the unsorted path outer union approach is created instead of the SQL for the sorted path outer union approach. The algorithm to generate the ordering sequence is the same.

7. Related Work

In this section, we discuss other work related to the content of this paper.

Middleware Queries for Producing XML Document Content: Fernandez et. al. [10] have evaluated the performance of different SQL query plans for generating XML document content. That work was done in the middleware context, with tagging done outside the database engine. Thus, the approaches proposed in [10] incur a significant data bind-out cost.

Fernandez et. al. also report experiments which show that the sorted outer union approach (with tagging done outside the engine) is not always optimal for generating the content of an XML document in the middleware. An execution strategy based on multiple SQL queries was shown to perform better in some cases. However, when we reran the same experiments using the DB2 database system, we found that the sorted outer union plan (generated using the view-tree reduction technique proposed in [10]) was always optimal. On further investigation, we learned that Fernandez et. al. used a different database engine, one in which null values were not compressed. This caused the size of the outer union plan results to be inflated by up to a factor of 3, thereby affecting sort performance. Therefore, we believe that the results presented in [10] regarding the performance of the sorted outer union approach do not apply to database engines that handle nulls efficiently (such as DB2). In these cases, the sorted outer union plan is likely to be optimal.

For database engines that do not handle null values efficiently, it is better to issue each individual leg of the sorted outer union plan (i.e., each path from the root to the leaves) as a separate SQL query. This corresponds to the fully partitioned strategy in [10] after the view-tree reduction step (the view-tree reduction step essentially avoids the need to issue a separate query for children that occur at most once per parent). Each of the queries issued in this manner is ordered by the ids of ancestors so that the constant-space tagger in the client can merge the results in a single pass over the SQL results. In this strategy, since there is no need to union the individual query results, there are no null values produced. Hence this strategy is (close to) optimal when null values are not handled efficiently by the database engine.

In contrast, when null values are handled efficiently by the database engine, the sorted outer union approach performs better for the following reasons. Firstly, the sorted outer union

approach avoids the overhead of issuing multiple queries. Secondly, the sorted outer union query can be optimized as a whole by the database engine. This enables the query-optimizer to do better memory management and exploit common sub-expression computation where applicable. Thirdly, the results of the individual legs of the sorted outer union are merged in document order by the database engine. This is better than the tagger at the client merging the individual legs because the database engine can do this merge more efficiently (using multiple disks, parallelism, buffering, etc.).

Query Languages for Publishing Relational Data as XML: In addition to the SQL extensions proposed in this paper, there are other language proposals for specifying the conversion from relational data to XML documents [2][11][15]. SilkRoute [11] uses a combination of SQL and XML-QL [7] (an XML query language), to specify the construction of XML documents. Microsoft Corporation uses XDR Schemas [15], which are annotated XML Schemas [27] for specifing the mapping from relational data to the desired XML document structure. Oracle Corporation uses object-relational types and object views to specify the structure of the desired XML document, and then uses a default tagging mechanism to publish a complex object as an XML document [2].

A distinguishing feature of our approach as compared to the SilkRoute and Microsoft approaches is that it extends SQL naturally, thus allowing the existing APIs (such as ODBC) and processing infrastructure of relational database systems to be reused. Further, our approach requires only simple extensions in the form of new SQL scalar and aggregate functions, which can easily be added to most existing relational database systems. In this sense our approach differs from Oracle's approach, which requires the relational database system to understand (and the user to create) sophisticated object-relational [25] types in order to publish relational data as XML documents.

Non-First Normal Form ($NF^{2)}$ Databases: The content of this paper is related to work on non-first normal form (NF^2) database systems. NF^2 database systems deal with the construction of nested relational tables, much like we deal with the construction of nested XML elements. There are, however, some key differences. The first difference is regarding the query language used to specify the construction of nested structures. While we naturally extend an existing query language (SQL), query languages proposed for NF^2 databases are either special-purpose ones, or semantics-modifying changes to existing query languages (such as SQL) [6][17][19]. Further, since NF^2 database systems do not deal with tags, their query languages cannot specify user-defined tagging of XML documents [14][20].

The work described in this paper also differs from the work on NF^2 database systems with respect to implementation techniques. One of the main reasons for this difference is that, unlike in NF^2 database systems, our focus is not on storing and querying nested objects – our focus is on creating and publishing nested objects from flat relations. Therefore, rather than developing storage and query-evaluation strategies over nested objects [6][20], we develop new strategies that can exploit a regular (flat) relational run-time mechanism to efficiently construct nested objects. Further, since tagging adds an extra dimension to the XML problem, we also develop new techniques to efficiently tag XML results.

Complex Object Assembly: The work on assembling complex object in object-oriented database systems [13][24] is also related to the content of this paper. Our work, however, differs from this work in that we exploit the sophisticated (set-oriented) query processing power of a relational query engine to construct complex XML elements from relational data.

Storing and Querying XML Documents using a RDBMS: There has been recent interest in using relational database systems to store and query XML documents [8][12][22]. The focus of this paper, however, is on efficiently publishing *existing relational data* as XML documents and addressing several of the key difficulties [22] encountered in that conversion.

8. Conclusion and Future Work

In this paper, we have studied ways to publish relational data in the form of structured XML documents. We proposed a SQL language extension (the XML constructor function) for specifying the construction of XML documents from relational data. By extending SQL in this manner, applications can reuse the vast infrastructure and APIs that exist today for SQL to extract XML documents from relational sources.

The bulk of this paper was devoted to exploring efficient mechanisms for publishing relational data as XML documents, independent of the actual language used to specify the outbound mapping. Towards this end, we first characterized the solution space based on the main differences between XML documents and relational tables, namely the presence of tags and nested structure. We then explored various alternatives in this space, paying special attention to the amount of processing that can be done inside the relational engine. Our experimental results show that moving all processing inside the relational engine can provide a significant performance benefit. This is because the high cost of binding out tuples to host variables is eliminated. Our study also shows that the outer union approaches that we have proposed in this paper provide an efficient and robust way to retrieve and structure the relational data needed to construct an XML document. In light of the superiority of the outer

union approaches, we have also presented algorithms to automatically generate the SQL queries for these approaches.

A number of possibilities exist for future work. These include studying the impact of parallelism, the addition of new runtime operators inside the relational engine to enhance the performance of outer union plans, and the design and analysis of techniques for efficient memory management to extend the useful range of the Unsorted Outer Union approach. In addition, we believe that the approaches outlined in this paper can be extended to handle the construction of recursive XML documents, such as part hierarchies and bill of material documents. Specifically, this last topic requires modifications to the tagger algorithms so that nested structures of arbitrary depth can be handled and also to the outer union approaches so that information about the unbounded hierarchy can be captured.

Acknowledgements

Amrish Lal implemented the initial version of the Correlated CLOB approach. Daniela Florescu and John Funderburk provided insightful comments on earlier drafts of this paper. Mary Fernandez, Atsuyuki Morishima and Dan Suciu provided us with a number of insights regarding the performance of the sorted outer union plan when null values are not handled efficiently by the database system. The VLDB 2000 conference referees also provided a number of helpful suggestions.

References

[1] American National Standards Institute, "The Database Language SQL", Standard No. X3.135-1992, New York, 1992.

[2] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, R. Murthy, "Oracle8i – The XML Enabled Data Management System", Proceedings of the International Conference on Data Engineering (ICDE), California, March 2000.

[3] D. Chamberlin, "A Complete Guide to DB2 Universal Database", Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.

[4] Commerce XML, http://www.cxml.org.

[5] R. Cover, The XML Cover Pages, http://www.oasis-open.org/cover/sgml-xml.html.

[6] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, G. Walch, "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies", Proceedings of the ACM SIGMOD Conference on the Management of Data, Washington D.C., May 1986.

[7] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, “XML-QL: A Query Language for XML”, Proceedings of the International World Wide Web (WWW) Conference, Toronto, May 1999.

[8] A. Deutsch, M. Fernandez, D. Suciu, “Storing Semi-Structured Data with STORED”, Proceedings of the ACM SIGMOD Conference on the Management of Data, Philadelphia, May 1999.

[9] R. Fagin, “Multi-valued Dependencies and a New Normal Form for Relational Databases”, ACM Transactions on Database Systems, 2(3), 1977.

[10] M. Fernandez, A. Morishima, D. Suciu, “Efficient Evaluation of XML Middle-ware Queries”, Proceedings of the ACM SIGMOD Conference on the Management of Data, Santa Barbara, California, May 2001 (to appear).

[11] M. Fernandez, W. Tan, D. Suciu, “SilkRoute: Trading Between Relations and XML”, Proceedings of the International World Wide Web (WWW) Conference, May 2000.

[12] D. Florescu, D. Kossman, “Storing and Querying XML Data using a RDBMS”, IEEE Data Engineering Bulletin, Vol. 22, No. 3, 1999.

[13] T. Keller, G. Graefe, D. Maier, “Efficient Assembly of Complex Objects”, Proceedings of the ACM SIGMOD Conference on Management of Data, Denver, Colorado, May 1991.

[14] H. F. Korth, M. A. Roth, “Query Languages for Nested Relational Databases”, Nested Relations and Complex Objects, Germany, April 1987.

[15] Microsoft Corporation, http://www.microsoft.com/xml.

[16] H. Pirahesh, J. Hellerstein, W. Hasan, “Extensible/Rule Based Query Rewrite Optimization in Starburst”, Proceedings of the ACM SIGMOD Conference on Management of Data, San Diego, California, June 1992.

[17] P. Pistor, F. Anderson, “Designing a Generalized NF^2 Model with an SQL-type Language Interface”, Proceedings of the Very Large Data Bases (VLDB) Conference, Kyoto, Japan, August 1986.

[18] Real Estate Transaction Standard, http://www.rets-wg.org.

[19] M. A. Roth, H. F. Korth, D. S. Batory, “SQL/NF: A Query Language for ¬NF Relational Databases”, Information Systems, 12(1), 1987.

[20] M. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, A. Verroust, “VERSO: A Database Machine Based On Nested Relations,” Nested Relations and Complex Objects”, Germany, April 1987.

[21] P. Seshadri, H. Pirahesh, T. Y. C. Leung, “Complex Query Decorrelation”, Proceedings of the International Conference on Data Engineering (ICDE), Louisiana, USA, February 1996.

[22] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang. D. DeWitt, J. Naughton, “Relational Databases for Querying XML Documents: Limitations and Opportunities”, Proceedings of the Very Large Data Bases (VLDB) Conference, Scotland, England, September 1999.

[23] L. D. Shapiro, “Join Processing in Database Systems with Large Main Memories”, ACM Transactions on Database Systems (TODS), Vol. 11, No. 3, 1986.

[24] E. Shekita, M. Carey, “A Performance Evaluation of Pointer-Based Joins”, Proceedings of the ACM SIGMOD Conference on the Management of Data, Atlantic City, New Jersey, June 1990.

[25] M. Stonebraker, D. Moore, P. Brown, “Object-Relational DBMSs: Tracking the Next Great Wave”, Morgan Kaufmann Publishers, September 1998.

[26] World Wide Web Consortium, “Extensible Markup Language (XML) 1.0 (Second Edition) ”, W3C Recommendation, October 2000. See http://www.w3c.org/TR/REC-xml.

[27] World Wide Web Consortium, “XML Schema Parts 0, 1, 2”, W3C Candidate Recommendation, October 2000. See http://www.w3c.org/TR/xmlschema-0, 1, 2.

[28] World Wide Web Consortium, “XQuery: A Query Language for XML”, W3C Working Draft, February 2001. See http://www.w3c.org/TR/xquery.