Efficiently Publishing Relational Data as XML Documents

Jayavel Shanmugasundaram* Eugene Shekita Rimon Barr^{+}
Michael Carey$^{\theta}$ Bruce Lindsay Hamid Pirahesh Berthold Reinwald

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95139
jai@cs.wisc.edu, shekita@almaden.ibm.com, barr@cs.cornell.edu,
carey@acm.org, bgl@almaden.ibm.com, pirahesh@almaden.ibm.com, 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.

*Also at the University of Wisconsin, Madison, WI 53706.

$^{+}$Work done at the IBM Almaden Research Center while the author was visiting from Cornell University, Ithaca, NY 14850.

$^{\theta}$Currently at Propel, 2350 Mission College Blvd., Santa Clara, CA 95054.

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

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 several industry proposals to standardize Document Type Descriptors (DTDs) [1], which are essentially schemas for XML documents. These DTDs are being developed for domains as diverse as electronic commerce [3] and real estate [10]. 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 fulfil its potential, some mechanism is needed to publish relational data in the form of XML documents.

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 scalar and aggregate function extensions for XML 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. Other recent proposals, based on a combination of SQL and XML query languages [8], do not share these advantages.

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.

1.1 Relationship to Related Work

There has been significant recent interest in using relational database systems to store and query XML documents [6][9][13]. The focus of this paper, however, is on efficiently publishing *existing relational data* as XML documents and addressing several of the key difficulties [13] in that conversion.

As mentioned earlier, there are other language proposals for specifying the construction of relational data as XML documents [8]. A distinguishing feature of our approach is that it extends SQL naturally, thus allowing the existing APIs and processing infrastructure of relational database systems to be reused. For example, the relational engine can be used to perform all join/merge operations during the construction of XML documents.

The content of this paper is related to work on set-valued attributes in object-relational databases [15] and nested non-first normal form data models [11]. They each deal with nested structures, much like we deal with nested XML elements. There are, however, some key differences. First, much of that work has been on special-purpose engines to process nested structures. In contrast, our goal is to ride on an underlying relational DBMS. Second, tagging adds an extra dimension to the XML problem that is not present in the O-R set world. Finally, our output is a static XML document rather than a structure accessible through nested cursors. This allows more optimizations to be performed inside the RDBMS.

1.2 Roadmap

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. We present our conclusions and ideas for future work in Section 6.

```
<customer id=”C1”>
   <name> John Doe </name>
   <accounts>
      <account id=”A1”> 1894654 </account>
      <account id=”A2”> 3849342 </account>
   </accounts>
   <porders>
      <porder id=”PO1” acct=”A1”>  // first purchase order
         <date>1 Jan 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>
      </porder>
      <porder id=”PO2” acct=”A2”>  // second purchase order
         …
      </porder>
   </porders>
</customer>
```

Figure 1: An XML Document Describing a Customer

2 An XML Primer

Extensible Markup Language (XML) [2] 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 [1] – 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

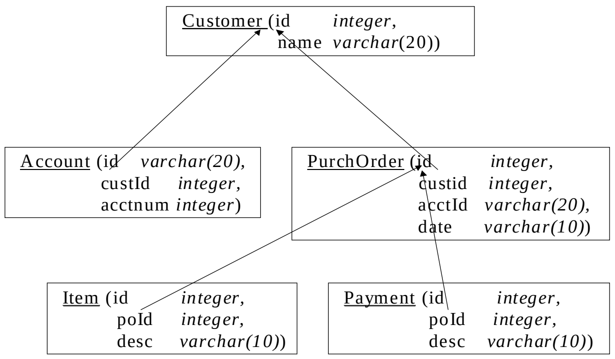

Figure 2: Customer Relational Schema

```
Define XML Constructor CUST (custId: integer,
                             custName: varchar(20),
                             acctList: xml,
                             porderList: xml) AS {
        <customer id=$custId>
                <name> $custName </name>
                <accounts> $acctList </accounts>
                <porders> $porderList </porders>
        </customer>
}
```

Figure 4: Definition of an XML Constructor

```
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.                          From    PurchOrder porder
13.                          Where  porder.custId = cust.id))
14. From  Customer cust
```

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

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 [2].

3 A SQL-Based Language Specification

A key requirement for converting relational data to XML documents is a language to specify the conversion. While one approach is to invent a new language specifically for this purpose [8], our approach is to harness and extend the power of SQL to specify the conversion of relational data to XML documents. 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 5-13). 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 1-13), 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 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. Since ordered inputs to aggregate functions are not currently supported in SQL, extensions similar to recent SQL amendments [4] would be necessary to make this possible.

This section has presented one possible language specification for converting relational data to XML documents. The rest of the paper is more general in scope – it examines different *implementations* to carry out the conversion, independent of the specification language.

4 Implementation Alternatives

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 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 physically tagging 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 explicitly (iteratively) issue a nested set of queries that matches the structure of the desired XML document. Consider the example shown in Figure 1. First a query can be issued to retrieve root level elements (customers). Information about a customer such as their

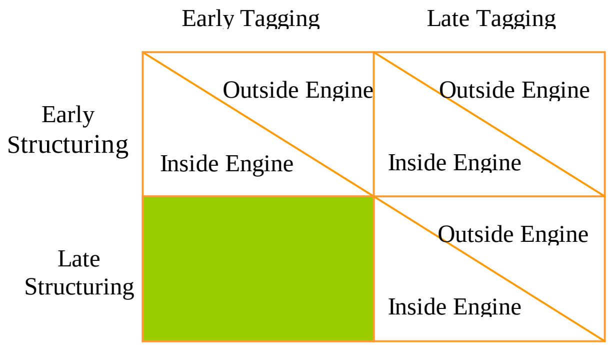

Figure 5: Space of Alternatives for Publishing XML

customer ID and customer name are retrieved, tagged, and output. 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. Then, for each purchase order retrieved, a separate query is 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. Note that nested structures in the XML document can be ordered using an "order by" clause in the issued SQL queries.

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. 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 dictates 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 queries SQL to the relational engine 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 the processing that 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

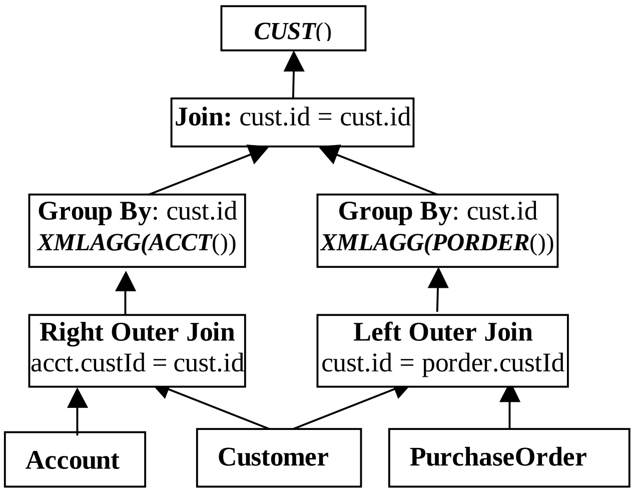

Figure 6: De-Correlated SQL Query with Aggregations

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. Since the XML document fragments created by the constructors 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.

Because of correlation during execution, the Correlated CLOB approach still performs a nested-loop join. It is likely to out-perform the Stored Procedure approach, however, 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 large objects are typically stored separately from the tuples they belong to. Thus, in parallel environments, fetching these objects (scattered around different nodes) can lead to significant performance degradation. Further, CLOBs may 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 *De-Correlated 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 [12] inside the relational engine to give the relational optimizer more flexibility. A de-correlated query execution plan for the correlated query of Figure 3 is shown in Figure 6. Though the Item and Payment tables are ignored for clarity, it is easy to see how this approach generalizes to arbitrary depths. First, each path from the root-level table to a leaf-level table is computed by joining the tables along the path (Customer joined with Account, Customer joined with Purchase Order). Outer joins are used because the information about a parent has to be preserved even if it has no children. The set of leaf-level XML elements corresponding to each leaf-level table is then built up (using aggregation) by grouping on the id columns of the parent tables on the path from the root-level table to the leaf-level table (e.g., custId). 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.

```
Select cust.*, acct.*, porder.*, pay.*, item.*
From  Customer cust
        left join Account acct on cust.id = acct.custId
        left join PurchOrder porder on cust.id = porder.custId
        left join Item item on porder.id = item.poId
        left join Payment pay on porder.id = pay.poId
```

Figure 7: Query for *Redundant Relation* Content

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 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: *Redundant Relation* Approach

One simple way to produce the needed content is to join all of the source tables. In our example, this would be done by joining the Customer, Accounts, Purchase Order, Item and Payment tables, as shown in Figure 7. Note that the join predicates relate parents to their children.

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 7 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 [7] 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.

4.2.2 Content Creation: *(Unsorted) 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 could be reduced. 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.

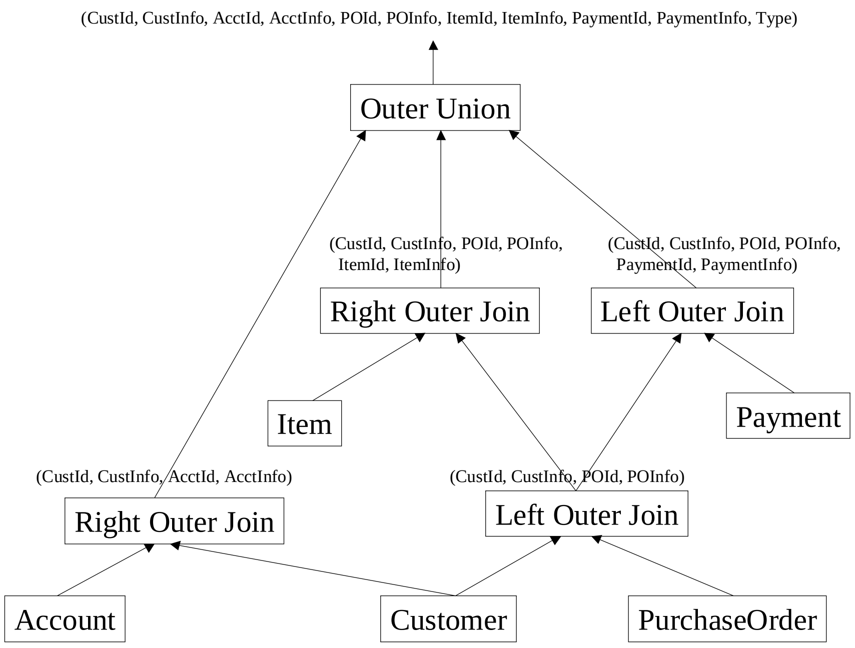
(CustId, CustInfo, AcctId, AcctInfo, POId, POInfo, ItemId, ItemInfo, PaymentId, PaymentInfo, Type)
Outer Union
(CustId, CustInfo, POId, POInfo, ItemId, ItemInfo)
(CustId, CustInfo, POId, POInfo, PaymentId, PaymentInfo)
Right Outer Join
Left Outer Join
Item
Payment
(CustId, CustInfo, AcctId, AcctInfo)
(CustId, CustInfo, POId, POInfo)
Right Outer Join
Left Outer Join
Account
Customer
PurchaseOrder

Figure 8: The Outer Union Plan

Figure 8 shows a query execution plan that reduces content redundancy for the query of Figure 3. First, as in the De-Correlated CLOB approach of Section 4.1.3, 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). This computation is shown in Figure 8 (see everything below the outer union). Where possible, common sub-expressions are used so that redundant computation is avoided. Thus, in Figure 8, the join between Customers and Purchase Orders is shared between two path computations.

Each path computation produces one tuple per data item in the leaf level of the XML tree. Each tuple describing a leaf level data item includes the information about all of its ancestors (see the lists of columns above each join box in Figure 8). A separate tuple describing an ancestor needs to be present only if the ancestor 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 (via the outer union in Figure 8) 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 need only four columns, while tuples representing items have six columns. In order to handle this heterogeneity, a separate column is allocated in the result of the union for each distinct column in the union's input. For each tuple representing a particular 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).

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 the approach exemplified by Figure 8 the *Path Outer Union* approach because it computes each *path* from the root-level table to a leaf-level table and *outer unions* them.

The Path Outer Union approach eliminates much of the data redundancy (and associated computation 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, parent information is replicated with every child of the parent (e.g., customer information is replicated with every account). One way to get around this is to feed the parent information directly into the outer union operator and to carry only parent ids along with the children. This reduces data redundancy, but it increases the number of tuples in the result because each parent is now represented by a separate tuple. We refer to this option as the *Node Outer Union* approach to distinguish it from the earlier Path Outer Union approach.

One concern with 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, with the remaining columns being null, in the absence of null value compression, this may lead to increased processing overhead due to larger tuple widths.

4.2.3 Structuring/Tagging: *Hash-based Tagger*

In the previous two 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 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, and this (single) aggregate function would logically perform the function of all the XML constructors and XMLAGGs in the user query. This ensures that large objects are not carried around during processing, which was 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 the 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). Thus, 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, 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, the parent is created and then the child is created. 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 can be written out as an XML file. If a specific order is required for the elements of the resulting XML document, 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 [14]. 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 in the hash-based tagger when memory is scarce. To eliminate this problem, the relational engine can be used to produce "structured content", which can then be tagged using a *constant space* tagger. We first explore a technique to produce structured content before describing the constant space tagger.

4.3.1 Structured Content Creation: *Sorted Outer Union* Approach

The key to structuring relational content is to order it the same 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 together.* This ensures that information about a particular node and its descendants is not mixed in with information about non-descendant nodes.
3) *The relative order of the tuples matches that of any user-specified order.* This is to handle user defined ordering requests.

We now show that performing a single final relational sort of the unstructured relational content is sufficient to ensure these properties. Our discussion here will be based on the Node (Unsorted) Outer Union approach for constructing unstructured relational content. The solution for the Path Outer Union Approach is actually simpler because it always satisfies condition 1. It is also easy to see how the technique generalizes to the Redundant Relation approach. In the interest of space, we only illustrate how the approach works when there is no user-specified ordering requirement (i.e., considering only conditions 1 and 2)

To ensure conditions 1 and 2, all that is required is to sort the result of the Node Outer Union on its id fields, with the ids of parent nodes occurring higher in the sort order than the ids of children nodes. Thus, in Figure 8, sorting the result on the composite key (CustId, AcctId, POId, ItemId, PaymentId) will ensure that result is in document order. 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). Condition 1 is then 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 ensure that tuples with null values in sort columns occur first, parent tuples (customers) will always occur before child tuples (accounts). Also, because the parent's id (customer id) occurs before a child's id (account id) in the sort order, the children of a parent node are grouped together after the parent, thus satisfying condition 2.

The Sorted Outer Union approach has the advantage of scaling to large data volumes because relational database sorting is disk-friendly. The approach can also ensure user-specified orderings with little additional cost. However, it does do more work than necessary, since a total order is produced when only a partial order is needed. This is because we only require children to occur together with parents and the ordering among siblings is immaterial (without user-specified ordering requirements).

Classification		Approach	Short Name
Early Tag *Early* Structure	Outside Engine	Stored Procedure	Stored Proc
	Inside Engine	Correlated CLOB	CLOB-Corr
	Inside Engine	De-Correlated CLOB	CLOB-DeCorr
Late Tag *Late* Structure	Inside or Outside Engine	Redundant Relation	Redundant R (In/Out)
	Inside or Outside Engine	Unsorted Path Outer Union	Unsorted OU (In/Out)
	Inside or Outside Engine	Unsorted Node Outer Union	Unsorted NOU (In/Out)
Late Tag *Early* Structure	Inside or Outside Engine	Sorted Path Outer Union	Sorted OU (In/Out)
	Inside or Outside Engine	Sorted Node Outer Union	Sorted NOU (In/Out)

Figure 9: Summary of Approaches for Publishing XML

4.3.2 Tagging Sorted Data: *Constant Space Tagger*

Once structured content is created, as described in the previous two sections, the next 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 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 now outlined a number of alternatives for creating XML documents from a relational database, which are summarized in Figure 9. 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 what situations). Our focus in this preliminary performance evaluation is to study the effects of nesting flat relational tables as nested XML documents. 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 effects of nesting relational data as XML documents, we will vary the nesting of the queries specifying 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 5-13 each have two directly nested sub-queries (lines 2-4, 5-13 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 queries in lines 2-4 and 5-13 and the third being queries in lines 6-8 and 9-12.

In our experiments, we only consider "balanced" queries, where 1) each non-leaf (sub) query has the same number of directly nested sub-queries and 2) 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 data for an XML element. Hence, for this performance study, the relational schema we use will reflect 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 (it is, however, important to note that each relation could, in general, be an arbitrarily complex view). Each table has an ID field, which is its primary key. It also has a PJID (parent join 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 PJID field of the parent and child tables, respectively. In addition to these two fields, each table has two data fields of different types. 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 database instance. The first parameter is *number of roots*, which specifies the number of tuples present in the table at the root level. The second parameter is the *number of leaf tuples*, which specifies the total number of tuples present in all 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 number of leaf tuples (where 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.

We now characterize the result XML document created for a given relational database instance. The integer and character column values of each tuple in the relational database instance are tagged as XML elements 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 effect 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. 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 implemented as local embedded-SQL programs, running on the same machine as the database server, to avoid unpredictable network delays. (We implemented "outside the engine" approaches as stored procedures as well, but since this did not significantly change their performance, these 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 Pentium 366 MHz 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 10: Parameter Settings for Experiments

For the experiments, we varied the parameters discussed in Section 5.1 as shown in Figure 10. For each experiment, we varied one of these parameters and used default values for the rest. This enabled us to determine the effect of each parameter on performance. Indexes were created on the ID and PJID fields for all 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 would be done in main memory; 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 Testing the Waters: Inside the Engine vs. Outside the Engine Approaches

To get an initial feel for the results, we first explore the effects of varying 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 11 and Figure 12, respectively. The Redundant Relation approach is not shown in these graphs because it performs very badly 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, at a query fan out of 4 was about 155 seconds. The performance of the Redundant Relation approach was also among the worst of all possible approaches throughout our experiments, so will not examine it further in our evaluation results.

The interesting thing to note in Figure 11 and Figure 12 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

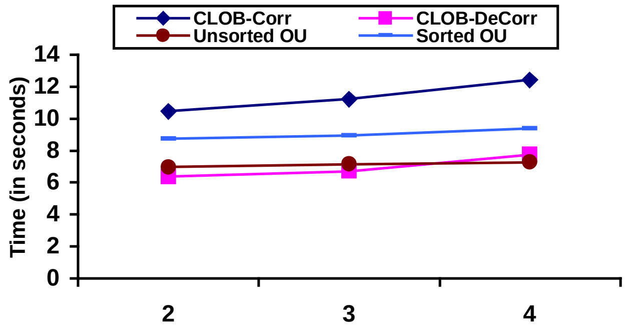

Figure 11: Varying Query Fan Out (Inside the Engine)

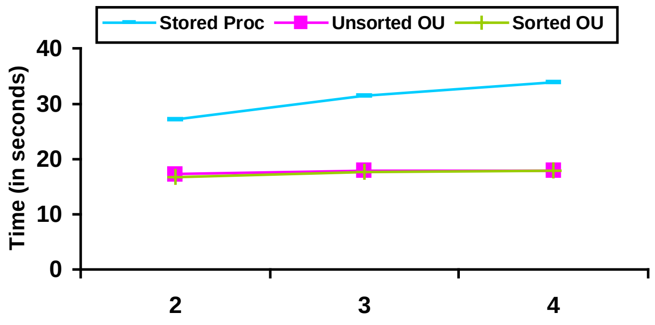

Figure 12: Varying Query Fan Out (Outside the Engine)

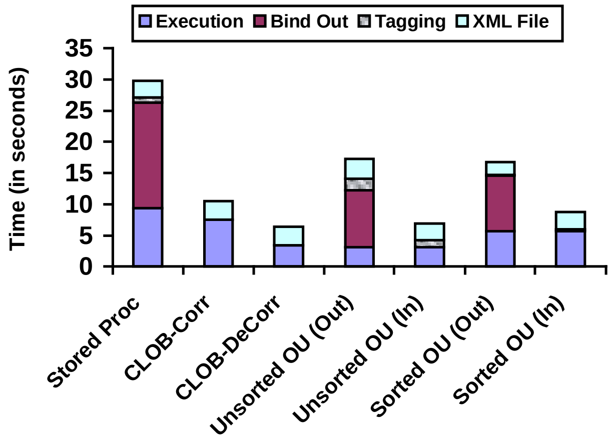

Figure 13: Break Down of XML Construction Time

"inside the engine" counterpart, takes less than 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 the corresponding "outside the engine" versions. In order to explain these results, we broke down the time for creating XML document results.

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 relational content to 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 there is no time spent in 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 out because it forms an integral part of the computation.

Figure 13 shows this time break down 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 to be true regardless of whether the bind out was done in a local client or in 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 give rise to much better performance. This points to our first firm conclusion – *constructing an XML document should be done inside the engine to maximize performance*.

Since "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, "outside the engine" approaches are valuable because they can be used with relational database systems that do not have support for the new XML scalar/aggregate functions mentioned in this paper.

5.4 Effect of Query Fan Out

We now re-examine 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 11. 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 plan whereby CLOBs could be retained in main memory without having to be materialized. Also, since

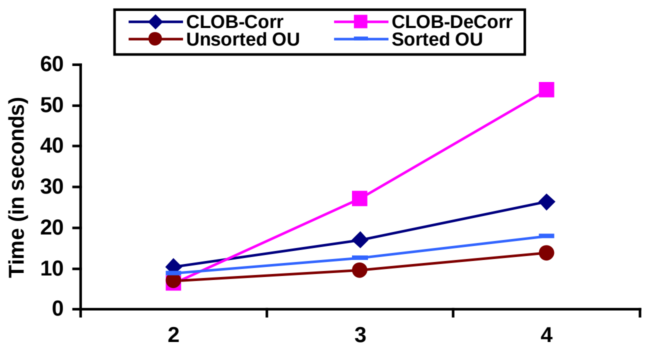

Figure 14: Varying Query Depth (Inside the Engine)

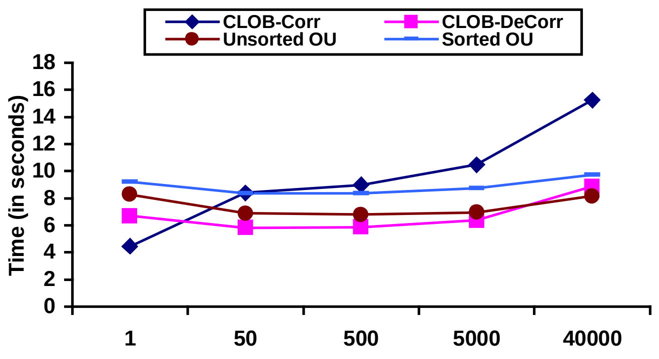

Figure 15: Varying Number of Roots (Inside the Engine)

the query depth is low, the overhead of repeatedly copying CLOBs is not significant.

Figure 12 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 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. 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 while 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 14 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 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 wrong 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 is compounded by the fact that the XMLAGG aggregate function is opaque to a traditional relational database optimizer and it thus has no good way to estimate the size of the CLOB result.

The effects of varying query depth for "outside the engine" approaches (not shown) are not very surprising, and essentially have the same form as Figure 12.

5.6 Effect of Number of Roots

The next parameter of interest is the number of roots. When the number of root elements is decreased for the "inside the engine" approaches, the performance of the Correlated CLOB approach improves dramatically, relative to the other approaches (see Figure 15). 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 (not shown). The relative performance of the outer union approaches remains unchanged.

5.7 Effects of Number of Leaf Tuples, Memory Size

For the next set of experiments, we varied the 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, 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, 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 spilling 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 an XML document inside the relational engine is far more efficient that doing so outside the engine, mainly because of the high cost of binding out tuples to host variables.
2) When processing can be done in main memory, a stable approach that is always among the very best (both inside and outside the engine), is the Unsorted Outer Union approach.
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.

6. Conclusion and Future Work

XML is rapidly emerging as the dominant standard for exchanging data on the World Wide Web, making the ability to publish data as XML increasingly important. 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) to specify the construction of XML documents from relational data. By extending SQL in this manner, applications can reuse the existing infrastructure and APIs 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 this mapping. Towards this end, we first characterized the solution space based on the main differences between XML documents and relational tables, namely 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 showed 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 showed that the outer union approaches proposed in this paper provide an efficient and robust way to retrieve the relational data needed to construct an XML document.

Possibilities for future work include studying the impact of parallelism, new runtime operators inside the relational engine to enhance the performance of outer union plans, and 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 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 using key columns.

7. Acknowledgements

Amrish Lal implemented the initial version of the Correlated CLOB approach. Daniela Florescu provided insightful comments on a draft of this paper.

8. References

[1] J. Bosak, et. al., “W3C XML Specification DTD,” http://www.w3.org/XML/1998/06/xmlspec-report.htm.

[2] T. Bray, J. Paoli, C. Sperberg-McQueen, “Extensible Markup Language (XML) 1.0,” http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm.

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

[4] “Database Language SQL. Amendment 1: On-Line Analytical Processing (SQL/OLAP)”, SC32 N00379, November 1999.

[5] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, “XML-QL: A Query Language for XML,” 8th International WWW Conference, Toronto, May 1999.

[6] A. Deutsch, M. Fernandez, D. Suciu, “Storing Semi-Structured Data with STORED,” SIGMOD Conference, Philadelphia, May 1999.

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

[8] M. Fernandez, W. Tan, D. Suciu, “SilkRoute: Trading Between Relations and XML,” 9th International WWW Conference, May 2000.

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

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

[11] M. Scholl, et. al., “VERSO: A Database Machine Based On Nested Relations,” Nested Relations and Complex Objects, Germany, April 1987.

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

[13] J. Shanmugasundaram, et. al., “Relational Databases for Querying XML Documents: Limitations and Opportunities,” Very Large Data Bases (VLDB) Conference, Scotland, September 1999.

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

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