BRIDGING RELATIONAL TECHNOLOGY AND XML

by

Jayavel Shanmugasundaram

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

2001

ABSTRACT

XML (eXtensible Markup Language) is fast emerging as the dominant standard for *representing* and *exchanging* information over the Internet. These two key applications of XML lead to exciting database opportunities. If data is stored and represented as XML documents, then it should be possible to query the contents of these documents in order to extract, synthesize and analyze their contents. *What is the best way to provide this query capability over XML documents?* On the other front, if XML is to serve as the medium for information exchange, then it should be possible to convert data stored in existing database systems into XML form for further processing by web-based applications. *How do we effectively publish data in existing database systems as XML documents?*

In this dissertation, we tackle these questions, focusing on the use of relational database technology to answer them. In particular, we make two contributions. First, we present a technique for storing and querying XML documents using a relational database system. This enables us to efficiently query XML documents by using a sophisticated, high-performance relational query-processor. Second, we propose and evaluate alternative strategies for efficiently publishing existing relational data as XML documents. These strategies enable us to publish XML documents for electronic-commerce applications, where, for the foreseeable future, most business data will almost certainly continue to be stored in relational database systems.

ACKNOWLEDGEMENTS

I would like to thank my academic advisor, Jeffrey Naughton, for his help and guidance throughout the course of this work. I have had a slightly unusual tenure as a graduate student, having spent the last two years working at the IBM Almaden Research Center in California. It has been largely due to Jeff's involvement and encouragement that I have been able to pursue (and successfully complete!) my dissertation research remotely.

I am fortunate to have spent two years as part of the database group at the University of Wisconsin. Much of my current understanding and appreciation of the database field in general has been due to my interactions with Professors David DeWitt, Jeffrey Naughton and Raghu Ramakrishnan. David and Jeff also pointed me in the direction of XML querying, which eventually led to this dissertation. I have also had a wonderful time working with the great bunch of database graduate students at Madison. While this group of people is too large to list here, I would like specially thank Kristin Tufte, Gang He and Chun Zhang for being such wonderful collaborators. Karthikeyan Ramasamy provided me with invaluable guidance regarding "hard-core" system issues.

The two years that I have spent at the IBM Almaden Research Center clearly ranks as one of the most intellectually stimulating periods in my life. The opportunity to work with world-class researchers and have an impact on an industrial-strength system has been both inspiring and satisfying. I would like to thank Michael Carey and Eugene Shekita for being my mentors during

my stay at Almaden. This dissertation would not have been possible without their guidance and supervision. I have also been lucky to work closely with a whole bunch of other folks at Almaden, including Kevin Beyer, Catalina Fan, John Funderburk, Jerry Kiernan, Sailesh Krishnamurthy, Bruce Lindsay, Chandrasekar Mohan, Hamid Pirahesh, Berthold Reinwald, Richard Sidle and Subbu Subramanian. Almaden has also been a fertile place for me to meet and work with "external collaborators", including Rimon Barr, Daniela Florescu, Zachary Ives, Rajasekar Krishnamurthy, Ying Lu, Igor Tatarinov, and Efstratios Viglas.

My interest in database research was sparked during my graduate school years at the University of Massachusetts at Amherst. Professors Lori Clarke, Barbara Lerner, Leon Osterweil and Krithi Ramamritham were all extremely supportive, and gave me all the freedom that I needed to pursue my interests. Krithi has also been a constant source of encouragement and advice throughout my graduate school years, both at Amherst and at Madison.

Life at Madison and San Jose would not have been as much fun without friends like Kevin Beyer, Abhinav Gupta, Jerry Kiernan, Arvind Nithrakashyap, Karthikeyan Ramasamy, Chandrasekar Sivasankaran and Sowmya Subramanian. This great bunch has helped keep me in good spirits throughout this process.

My parents and sister have been my greatest source of support, and I would not have been able to complete this dissertation without their encouragement. I would thus like to dedicate this dissertation to them. I would also like to thank my nephew, Rupin Krishnan, for giving me a much-needed ego-boost every once in a while ☺.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

XML has the potential to revolutionize the Internet. Like HTML, XML provides a tagged, hierarchical format for representing documents. However, while the primary purpose of HTML tags is to describe how a data item is to be displayed, XML tags describe the data itself. The importance of this simple distinction cannot be underestimated – because XML data is self-describing, it becomes possible to represent information without loss of semantics in XML documents and it also becomes possible for programs to interpret this information. Thus, an application receiving an XML document can interpret it in multiple ways, can filter the document based on its content, can restructure it to suit the application's needs, and so on. Consequently, this self-describing feature of XML, coupled with its nested structure that naturally models complex objects, makes it ideal for both *representing* and *exchanging* information over the Internet.

These two key applications of XML lead to exciting database opportunities. If documents are stored and represented in XML form, then it should be possible to query the contents of these documents to extract, synthesize and analyze their contents. For example, if bibliography listings are represented as XML documents, it should, for instance, be possible to find all publications authored by Charles Darwin. *What is the best way to provide this query capability over XML documents?* On the other front, if XML is to serve as the medium for information exchange, then we need to be able to tap the vast volumes of important data stored in existing database systems and convert them into XML form for further processing by web-based applications. For example, business-to-business application servers can retrieve data from backend relational databases to produce

inventory lists and purchase orders in XML form and send them to business partners. This issue assumes special significance in the context of electronic commerce applications because, for the foreseeable future, most business data will almost certainly continue to reside in relational database systems. *How do we effectively publish such data in existing database systems as XML documents?* This dissertation is devoted to tackling these two important questions, concentrating specifically on the use of relational database technology to solve these problems.

1.1 Efficiently Querying XML Documents

When one asks what the best way is of providing a query capability over XML documents, the answer seems obvious. Since an XML document is a prime example of semi-structured data (it is tree structured with self-describing tags), why not just use semi-structured query languages and query evaluation techniques? This is indeed a viable alternative and there is considerable work in the semi-structured database community focused on exploiting this approach. The question we ask here, however, is whether this is the best or the only approach to take. The down side of using semi-structured databases is that they ignore nearly three decades of research and development in building and maturing commercial strength relational databases systems. Can we not reuse existing relational database technology for querying data represented as XML documents?

The good news is that this is possible. The key feature that makes it possible is the presence of Document Type Descriptors (DTDs), which are essentially loose schemas for XML documents. Our approach is to have a translation layer on top of a commercial relational database management system that can (a) convert a DTD to a relational schema, (b) store XML documents

conforming to DTDs as tuples, (c) translate queries over the XML documents to SQL queries and (d) convert the relational results back to XML form.

While the translation layer approach clearly works, during the course of its implementation we have identified certain limitations of the relational model and of current relational engines that make some queries awkward or inefficient to process. We thus propose extensions that can make relational database systems more effective for querying XML documents. While some of these extensions have been proposed in other contexts, one of the issues to be addressed is specific to XML and crucial to the success of any relational technology based XML query system – that of efficiently constructing nested, tagged XML results from flat relational tables. Not surprisingly, this is also a central issue in our quest to publish existing relational data as XML documents for the purpose of information exchange in the Internet and so we will discuss this issue further in that context.

1.2 Publishing Relational Data as XML Documents

A general and flexible way to publish relational data as XML documents is to create (possibly many) XML views of the underlying relational data. Each of these XML views can provide an alternative, application-specific view of the underlying relational data. Through these XML views, XML application developers can access relational data as though it was in some standard XML format.

There are two main challenges that arise in supporting XML views of relational data. The first is to efficiently materialize the contents of an XML view as a nested XML document. This is important because relational data has to be materialized in XML form for further processing by XML

applications. The second challenge is to support query capability over XML views of relational data. This is important because, in many cases, applications do not require all the contents of an XML view to be materialized. For example, in an XML view of available items, an application may only be interested in a particular item at a given point in time. Materializing all available items would be wasteful in this case because it would result in unnecessary computation. A better solution is to support queries over XML views so that application developers can retrieve only the data items of interest. Supporting queries over XML views also allows application developers to synthesize data from different XML views. We now consider these issues in more detail.

1.2.1. Efficiently Materializing Relational Data as XML Documents

There are many challenges involved in materializing relational data as XML documents. This is because relations 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 have characterized the space of alternatives based on whether tagging and structuring are done early or late during query processing. We have also refined this space based on how much processing is done inside the relational engine and how much is done outside. Based on this characterization, we have explored various alternatives, classifying existing techniques and proposing some new ones. Our performance comparison of the various alternatives shows that one of the new techniques, based on late structuring and late tagging, is attractive when the result XML document fits in memory; another technique, based on early structuring and late tagging, performs well otherwise. Our results also indicate that constructing an XML document inside the relational engine is far more efficient than doing so outside. This latter result has interesting implications. It suggests that

extending relational technology for the purpose of XML is likely to have a significant performance payoff.

1.2.2. Querying XML Views of Relational Data

In the context of querying XML views of relational data, we have focused on two issues. The first is the design of a general framework for processing arbitrarily complex XML queries over XML views, including queries with features such as nested expressions and nested order. The other area of focus is performance, whereby we present techniques for efficiently evaluating XML queries over XML views of relational data. One such technique is XML view composition, which eliminates the construction of all intermediate XML fragments that do not appear in the final query result. Another performance-enhancing technique is what we call "computation push-down". This pushes all data and memory intensive computation in an XML query down to the relational engine. As a result, the query processing power of a relational engine is used to efficiently evaluate XML queries. Only a small memory-efficient tagger is required outside the relational engine to tag the SQL results and produce the resulting XML.

Based on our implementation of the above techniques, we have identified some of the limitations that arise from using a relational query processor for executing XML queries. These limitations are due to the semantic mismatch between an XML query language and SQL. We have identified the causes for this mismatch and we propose possible solutions to help overcome this problem.

1.3 Outline of this Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we present an overview of XML and related technologies. In Chapter 3, we present our technique for storing and query XML documents using a relational database system. In Chapter 4, we propose and evaluate techniques for efficiently materializing relational data as XML documents. In Chapter 5, we present techniques for efficiently querying XML views of relational data. In Chapter 6, we present related work, and in Chapter 7, we present our conclusions and outline avenues for future research.

CHAPTER 2: XML AND RELATED TECHNOLOGIES

In this chapter, we provide some background on XML and other technologies that are related to the content of this dissertation. Many of these technologies are evolving standards and this chapter provides a snapshot that is up-to-date as of July 2001.

The remainder of this chapter is organized as follows. In Section 2.1, we gives an overview of the XML standard. In Section 2.2 we describe DTDs, which are used to describe the schema of XML documents. In Section 2.3, we describe various XML query language proposals.

2.1. The eXtensible Markup Language (XML)

The eXtensible Markup Language (XML) [74] is a hierarchical format for information representation and exchange over the Internet. 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

```
<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 Example XML Document

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, and it logically points to an element having the same value as its ID (such IDREF typing information is specified in a Document Type Descriptor associated with an XML document, as described in the next section). 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 [74].

2.2. Document Type Descriptors (DTDs)

Document Type Descriptors (DTDs) [74] describe the structure of XML documents and are like a schema for XML documents. A DTD specifies the structure of an XML element by specifying the names of its sub-elements and attributes. Sub-element structure is specified using the operators * (set with zero or more elements), + (set with one or more elements), ? (optional), and | (or). All values are assumed to be string values unless the type is ANY, in which case the value can be an arbitrary XML fragment. Each element can also have many attributes. There is a special attribute of type ID, which can occur once for each element. The id attribute uniquely identifies an element within a document and can be referenced through an IDREF attribute in another element. IDREFs are untyped in the sense that they can point to the id field of any element. There is no concept of a root of a DTD – an XML document conforming to a DTD can be rooted at any element specified in the DTD.

Figure 2 shows an example DTD specification, which specifies the schema for a class of XML documents including the one shown in Figure 1. As shown in the DTD specification, a *customer* element has *name*, *account* and *porders* sub-elements (line 1). The *name* sub-element is optional, as specified by the "?" in the DTD. In addition, a *customer* element has an id attribute that is of type ID (line 2). The #REQUIRED annotation specifies that the attribute has to be present for every *customer* element.

The sub-elements of the *customer* element are defined similarly. For example, the *name* sub-element has no attributes, and has a text value specified as #PCDATA (line 3). The *accounts* sub-element has one or more *account* sub-elements (line 4). Each of these *account* sub-elements in turn has a text

```
 1. <!ELEMENT customer (name?, accounts, porders)>
 2. <!ATTLIST    customer  id ID #REQUIRED)>
 3. <!ELEMENT name (#PCDATA)>
 4. <!ELEMENT accounts (account+)>
 5. <!ELEMENT account (#PCDATA)>
 6. <!ATTLIST    account  id ID #REQUIRED>
 7. <!ELEMENT porders (porder*)>
 8. <!ELEMENT porder (date, items, payments)>
 9. <!ATTLIST    porder  ID #REQUIRED
10.                 acct      IDREF #IMPLIED>
11. <!ELEMENT date (#PCDATA)>
12. <!ELEMENT items (item+)>
13. <!ELEMENT item (#PCDATA)>
14. <!ATTLIST    item  ID #REQUIRED>
15. <!ELEMENT payments (payment*)>
16. <!ELEMENT payment (#PCDATA)>
17. <!ATTLIST    payment  ID #REQUIRED>
```

Figure 2: An Example DTD

value (line 5) and an id attribute (line 6). The *porder* sub-elements of a *customer* element and its descendants are defined similarly.

2.3. XML Query Languages

There have been many languages proposed for querying XML documents [3][11][13][27][76][79]. All of these query languages have the notion of path expressions for navigating the nested structure of XML documents. In addition, some of them have element constructors for creating nested XML elements. In this section, we briefly introduce three query languages – Lorel [3], XML-QL [27] and XQuery [79] – that we shall use for the rest of this dissertation. Each of these three query languages has some features that we exploit and that are not found in the other two. The following description of the query languages is only intended to provide the reader with a high-level overview. Additional details will be presented as necessary in the subsequent chapters.

```
Select  C.name
From    customer C
Where   C.porders.porder.items.item = "Bungee Ropes"
```

Figure 3: An Example Lorel Query

Let us consider a query over a set of XML documents (such as the XML document in Figure 1) that selects the names of all customers who have bought "Bungee Ropes". The Lorel version of the query is shown in Figure 3. As shown, the Lorel query is SQL-like in that it has "select", "from" and "where" clauses. The "from" clause binds to all customer elements, and the "where" clause selects only those that have an item with value "Bungee Ropes". This is specified as a predicate in which a path expression (C.porders.porder.items.item) is used to pull out all the items associated with a customer. It is important to note that predicates in Lorel have an implicit existential semantics. In our example, the predicate evaluates to true if *any* item associated with a customer is a "Bungee Rope". The "select" clause returns the names of the selected customers.

Figure 4 shows the same query written using XML-QL. As shown, there is a "where" part that binds to the desired XML elements, and a "construct" part that formats the output result. In our example, the "where" part binds the variable $custname to the names of those customers that have bought an item having the value "Bungee Ropes". The "construct" part tags the selected $custname variables under the newly constructed XML element having the tag name "brave_customer". This syntax can be used to create arbitrarily nested XML structures in XML-QL (Lorel does not have this capability to create new XML element structures in the output).

Figure 5 shows our running example written using XQuery. An XQuery FLWR expression (short for For-Let-Where-Return expression) is used to create and construct the result customer

```
WHERE <customer>
            <name> $custname </name>
            <porders>
               <porder>
                  <items>
                     <item> Bungee Ropes </item>
                  </item>
               </porder>
            </porders>
         </customer>
CONSTRUCT <brave_customer> $custname </brave_customer>
```

Figure 4: An Example XML-QL Query

```
for $cust in /customer
where $cust/porders/porder/items/item = "Bungee Ropes"
return <brave_customer> $cust/name </brave_customer>
```

Figure 5: An Example XQuery Query

elements. First, each customer is bound to the variable $cust (line 1). Second, only those customers that have bought "Bungee Ropes" are selected (line 2). This is done by navigating to the items bought by a customer ($cust/porders/porder/items/item) using XPath [76] syntax, and then seeing whether any of them have the value "Bungee Ropes". Finally, the names of the selected customer are output under a newly created XML element.

CHAPTER 3: QUERYING XML DOCUMENTS USING A RELATIONAL DATABASE SYSTEM

In this chapter, we demonstrate that it is possible to use a standard commercial relational database system to evaluate powerful queries over XML documents. The key that makes this possible is the existence of schema information, such as DTDs [74] or XML Schemas [77], for XML documents. Without such schema information, XML will never reach its full potential because a tagged document is not very useful without some agreement among inter-operating applications as to what the tags mean. Put another way, the reason the Internet community is so excited about XML is that there is the vision of a future in which the vast majority of files on the web are XML files conforming to some standardized schema. An application encountering such a file can interpret the file by consulting the schema to which the document conforms.

Our approach to querying XML documents is as follows. First, we process a DTD to generate a relational schema. Second, we parse XML documents conforming to DTDs and load them into tuples of relational tables in a standard commercial relational database system (in our case, this is IBM's DB2 database [18]). Third, we translate semi-structured queries (specified in a language similar to XML-QL [27] or Lorel [3]) over XML documents into SQL queries over the corresponding relational data. Finally, we convert the results back to XML. This high-level architecture is shown in Figure 6.

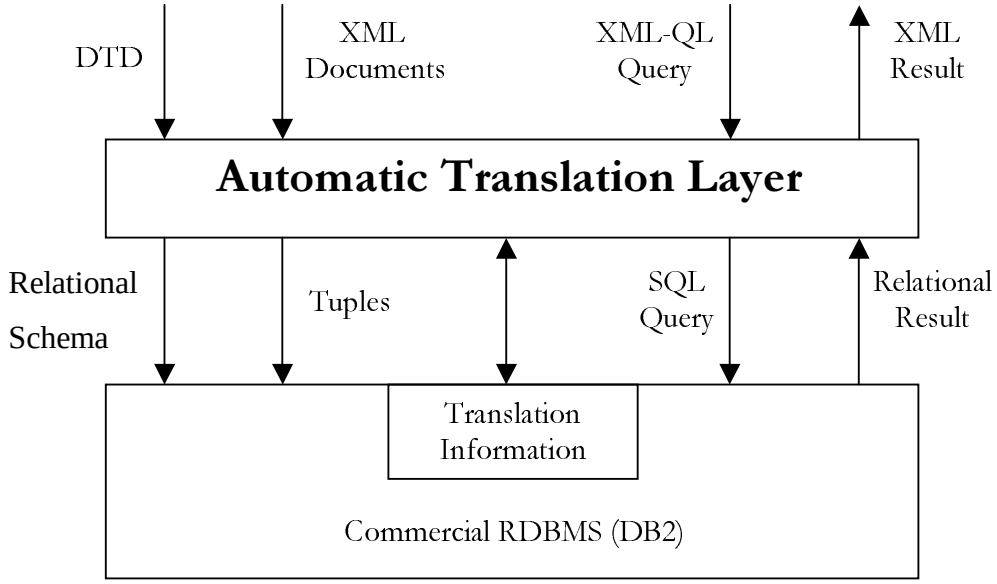

Figure 6: High-level Query Processing Architecture

The good news is that this works. A main contribution of this chapter is the description of an approach that enables one to take the XML queries, data sets, and schemas so foreign to the relational world and process them in relational systems without any manual intervention. This means that we are presented with a large opportunity: all of the power of relational database systems can be brought to bear upon the XML query problem.

However, the fact that something is possible does not necessarily imply that it is a good idea. Our experience with implementing this system and using it with over 30 different XML DTDs has revealed that there are a number of limitations in current relational database systems that in some instances make using relational technology for XML queries either awkward or inefficient. Relational technology proves awkward for queries that require complex XML constructs in their results, and may be inefficient when fragmentation due to the handling of set-valued attributes and sharing causes too many joins in the evaluation of simple queries. Another contribution of this chapter is the identification of those limitations, and a discussion of how they may be removed. It

is an open question at this point whether the best approach is to start with relational technology and try to remove these limitations, or to start with a semi-structured system and try to add the power and sophistication currently found in relational query processing systems.

The remainder of this chapter is organized as follows. Section 3.1 presents the algorithms for translating DTDs and XML documents to a relational format, and also presents an evaluation of these algorithms using real DTDs. Section 3.2 Describes the translation of queries over XML documents to SQL queries. Section 3.3 deals with the conversion of the results to XML. Section 3.4 proposes extensions to the relational model that will make it more suitable for processing XML documents.

3.1. Storing XML Documents in a Relational Database System

In this section, we describe how to generate relational schemas from XML DTDs. The main issues that must be addressed include: (a) dealing with the complexity of DTD element specifications, (b) resolving the conflict between the two-level nature of relational schemas (table and attribute) vs. the arbitrary nesting of XML DTD schemas, and (c) dealing with set-valued attributes and recursion.

3.1.1. Simplifying DTDs

In general, DTDs can be complex and generating relational schemas that capture this complexity would be unwieldy at best. Fortunately, one can simplify the details of a DTD and still generate a relational schema that can store and query documents conforming to that DTD. Note that it is not necessary to be able to regenerate a DTD from the generated relational schema. Rather, what is

required is that (a) any document conforming to the DTD can be stored in the relational schema, and (b) any XML semi-structured query over a document conforming to the DTD can be evaluated over the relational database instance.

Most of the complexity of DTDs stems from the complex specification of the type of an element. For instance, we could specify an element "a" as <!ELEMENT a ((b|c|e)?,(e?|(f?,(b,b)*))*)>, where "b", "c", "e" and "f" are other elements. However, at the query language level, all that matters is the position of an element in the XML document relative to its siblings, and the parent-child relationship between elements in the XML document. We now propose a set of transformations that can be used to "simplify" any arbitrary DTD without undermining the effectiveness of queries over documents conforming to that DTD. These transformations are a superset of the ones presented in [28].

The transformations are of three types: (a) flattening transformation, which convert a nested definition into a flat representation (i.e., one in which the binary operators "," and "|" do not appear inside any operator – see Figure 7), (b) simplification transformations, which reduce many unary operators to a single unary operator (Figure 8), and (c) grouping transformations, which group sub-elements having the same name (for example, two a* sub-elements are grouped into one a* sub-element – see Figure 9). In addition, all "+" operators are transformed to "*" operators. Our example specification would be transformed to: <!ELEMENT a (b*, c?, e*, f*)>.

The transformations preserve the semantics of: (a) one or many, and (b) null or not null. The astute reader may notice that we have lost information about relative orders of the elements. This is true; fortunately, this information can be captured when a specific XML document is loaded into

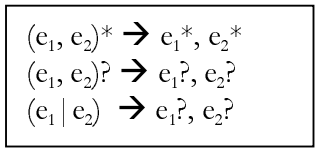

Figure 7: Flattening Transformations

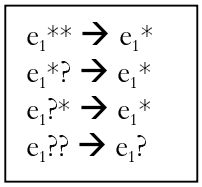

Figure 8: Simplification Transformations

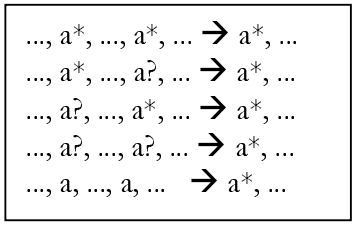

Figure 9: Grouping Transformations

this relational schema (e.g., by position fileds in the tuples representing some of the elements). We now explore techniques for converting a simplified DTD to a relational schema.

3.1.2. Motivation for Special Schema Conversion Techniques

Traditionally, relational schemas have been derived from a data model such as the Entity-Relationship model. This translation is relatively straightforward because there is a clear separation between entities and their attributes. Each entity and its attributes are mapped to a separate relation.

When converting an XML DTD to relations, it is tempting to map each element in the DTD to a relation and map the attributes of the element to attributes of the relation. However, there is no correspondence between elements and attributes of DTDs and entities and attributes of the ER-Model respectively. What would be considered "attributes" in an ER-Model are often most naturally represented as elements in a DTD. Figure 10 shows a DTD that illustrates this point. This DTD describes the schema of XML documents representing publications, their titles and authors. In an ER-Model, *author* would be an "entity" and *firstname*, *lastname* and *address* would be

```
<!ELEMENT book (booktitle, author)>

<!ELEMENT article (title, author*, contactauthor)>

<!ELEMENT contactauthor EMPTY>
<!ATTLIST    contactauthor authorID IDREF IMPLIED>

<!ELEMENT monograph (title, author, editor)>

<!ELEMENT editor (monograph*)>
<!ATTLIST    editor name CDATA #REQUIRED>

<!ELEMENT author (name, address)>
<!ATTLIST    author id ID #REQUIRED>

<!ELEMENT name (firstname?, lastname)>

<!ELEMENT firstname (#PCDATA)>

<!ELEMENT lastname (#PCDATA)>

<!ELEMENT address ANY>
```

Figure 10: An Example DTD

attributes of that entity. In designing the DTD, however, there is no incentive to make *author* an element and *firstname*, *lastname* and *address* attributes. In fact, in XML, if *firstname* and *lastname* were attributes, they could not be nested under name as is done in the DTD (because XML attributes cannot have a nested structure). Directly mapping elements to relations is thus likely to lead to excessive fragmentation of the document, which in turn would translate to many joins in order to execute queries.

3.1.3. The Basic Inlining Technique

The Basic Inlining Technique, hereafter referred to as *Basic*, solves the fragmentation problem by inlining as many descendants of an element as possible into a single relation. However, *Basic* creates relations for every element because an XML document can be rooted at any element in a

DTD. For example, the *author* element in Figure 10 would be mapped to a relation, with attributes *firstname*, *lastname* and *address*. In addition, relations would be created for *firstname*, *lastname* and *address*.

We must address two complications: set-valued attributes and recursion. In the example DTD in Figure 10, when creating a relation for article, we cannot inline the set of authors because the traditional relational model does not support set-valued attributes. Rather, we follow the standard technique for storing sets in an RDBMS, and create a separate relation for author and link authors to articles using a foreign key. Just using inlining (if we want the process to terminate) necessarily limits the level of nesting in the recursion. Therefore, we express the recursive relationship using the notion of relational keys and use relational recursive processing to retrieve the relationship. In order to do this in a general fashion, we introduce the notion of a DTD graph.

A DTD graph represents the structure of a DTD. Its nodes are elements, attributes and operators in the DTD. Each element appears exactly once in the graph, while attributes and operators appear as many times as they appear in the DTD. The DTD graph corresponding to the DTD in Figure 10 is given in Figure 11. Cycles in the DTD graph indicate the presence of recursion.

The schema created for a DTD is the union of the sets of relations created for each element. In order to determine the set of relations to be created for a particular element, we create a graph structure called the *element graph*. The element graph is constructed as follows.

Do a depth first traversal of the DTD graph, starting at the element node for which we are constructing relations. Each node is marked as "visited" the first time it is reached and is

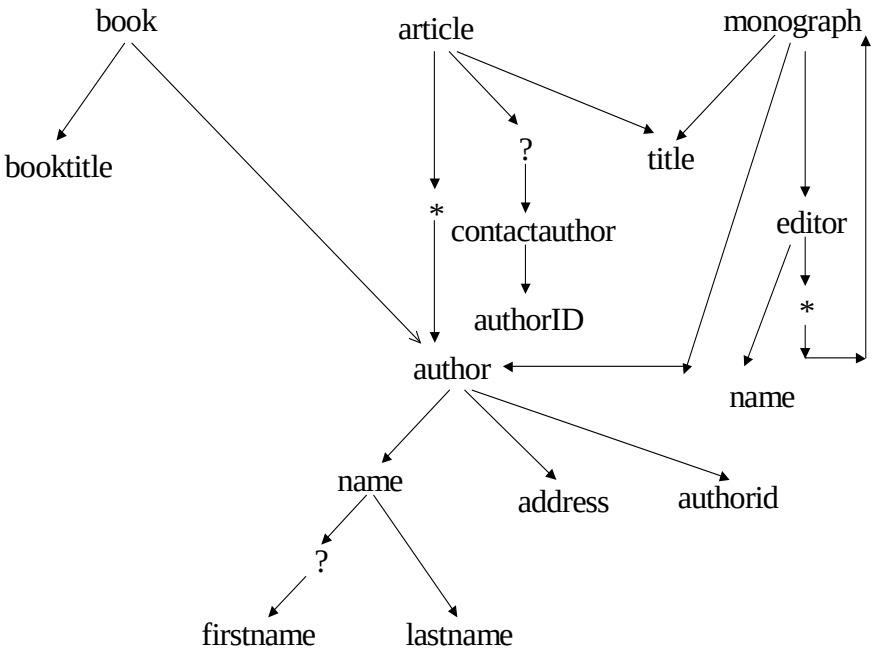

Figure 11: A DTD Graph

unmarked once all its children have been traversed. If an unmarked node in the DTD graph is reached during depth first traversal, a new node bearing the same name is created in the element graph. In addition, a *regular* edge is created from the node created for the DFS parent of the current DTD node to the newly created node. If an attempt is made to traverse an already marked DTD node, then a *backpointer* edge is added from the node created for the DFS parent of the current DTD node to the most recently created node in the element graph with the same name as the marked DTD node.

The element graph for the *editor* element in the DTD graph in Figure 11 is shown in Figure 12. Intuitively, the element graph expands the relevant part of the DTD graph into a graph structure.

Given an element graph, relations are created as follows. A relation is created for the root element of the graph. All the element's descendants are inlined into that relation with the following two

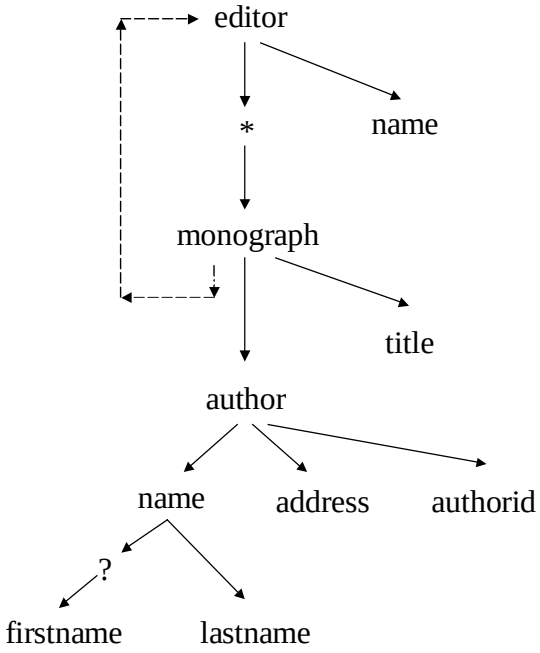

Figure 12: An Element Graph

exceptions: (a) children directly below a "*" node are made into separate relations – this corresponds to creating a new relation for a set-valued child, and (b) each node having a backpointer edge pointing to it is made into a separate relation – this corresponds to creating a new relation to handle recursion.

Figure 13 shows the relational schema that would be generated for the DTD in Figure 10. There are several features to note in the schema. Attributes in the relations are named by the path from the root element of the relation. Each relation has an ID field that serves as the key of that relation. All relations corresponding to element nodes having a parent also have a parentID field that serves as a foreign key. For instance, the *article.author* relation has a foreign key *article.author.parentID* that joins authors with articles. As an example of shredding an XML

book (bookID: integer, book.booktitle : string, book.author.name.firstname: string,
book.author.name.lastname: string, book.author.address: string, author.authorid: string)
booktitle (booktitleID: integer, booktitle: string)
article (articleID: integer, article.contactauthor.authorid: string, article.title: string)
article.author (article.authorID: integer, article.author.parentID: integer, article.author.name.firstname: string,
article.author.name.lastname: string, article.author.address: string, article.author.authorid: string)
contactauthor (contactauthorID: integer, contactauthor.authorid: string)
title (titleID: integer, title: string)
monograph (monographID: integer, monograph.parentID: integer, monograph.title: string,
monograph.editor.name: string, monograph.author.name.firstname: string,
monograph.author.name.lastname: string, monograph.author.address: string,
monograph.author.authorid: string)
editor (editorID: integer, editor.parentID: integer, editor.name: string)
editor.monograph (editor.monographID: integer, editor.monograph.parentID: integer,
editor.monograph.title: string, editor.monograph.author.name.firstname: string,
editor.monograph.author.name.lastname: string,
editor.monograph.author.address: string, editor.monograph.author.authorid: string)
author (authorID: integer, author.name.firstname: string, author.name.lastname: string, author.address: string,
author.authorid: string)
name (nameID: integer, name.firstname: string, name.lastname: string)
firstname (firstnameID: integer, firstname: string)
lastname (lastnameID: integer, lastname: string)
address (addressID: integer, address: string)

Figure 13: Relational Schema Generated using *Basic*

document into these relations, the XML document in Figure 14 would be converted to the following tuple in the book relation:

(1, The Selfish Gene, Richard, Dawkins, <city>Timbuktu</city><zip>99999</zip>, dawkins).

The ANY field, address, is stored as an uninterpreted string; thus the nested structure is not visible to the database system without further support for XML (see Section 3.4). Note that if the author Richard Dawkins has authored many books, then the author information will be replicated for each book because it is replicated in the corresponding XML documents.

While *Basic* is good for certain types of queries, such as "list all authors of books", it is likely to be grossly inefficient for other queries. For example, queries such as "list all authors having first name

```
<book>
   <booktitle> The Selfish Gene </booktitle>
   <author id = "dawkins">
      <name>
         <firstname> Richard </firstname>
         <lastname> Dawkins </lastname>
      </name>
      <address>
         <city> Timbuktu </city>
         <zip> 99999 </zip>
      </address>
   </author>
</book>
```

Figure 14: An Example XML Document

Jack" will have to be executed as the union of 5 separate queries. Another disadvantage of *Basic* is the large number of relations it creates. Our next technique attempts to resolve these problems.

3.1.4. The Shared Inlining Technique

The Shared Inlining Technique, hereafter referred to as *Shared*, attempts to avoid the drawbacks of *Basic* by ensuring that an element node is represented in exactly one relation. The principal idea behind *Shared* is to identify the element nodes that are represented in multiple relations in *Basic* (such as the *firstname*, *lastname* and *address* elements in the example) and to share them by creating separate relations for these elements.

We must first decide what relations to create. In *Shared*, relations are created for all elements in the DTD graph whose nodes have an in-degree greater than one. These are precisely the nodes that are represeted as multiple relations in *Basic*. Nodes with an in-degree of one are inlined. Element nodes having an in-degree of zero are also made separate relations, because they are not reachable from any other node. As in *Basic*, elements below a "*" node are made into separate relations. Finally, of the mutually recursive elements all having in-degree one (such as *monograph* and *editor* in

book (bookID: integer, book.booktitle.isroot: boolean, book.booktitle : string)
article (articleID: integer, article.contactauthor.isroot: boolean, article.contactauthor.authorid: string)
monograph (monographID: integer,monograph.parentID: integer, monograph.parentCODE: integer,
monograph.editor.isroot: boolean, monograph.editor.name: string)
title (titleID: integer, title.parentID: integer, title.parentCODE: integer, title: string)
author (authorID: integer, author.parentID: integer, author.parentCODE: integer, author.name.isroot: boolean,
author.name.firstname.isroot: boolean, author.name.firstname: string, author.name.lastname.isroot: boolean,
author.name.lastname: string, author.address.isroot: boolean, author.address: string, author.authorid: string)

Figure 15: Relational Schema Generated using Shared

Figure 11), one of them is made a separate relation. We can find such mutually recursive elements by looking for strongly connected components in the DTD graph.

Once we decide which element nodes are to be made into separate relations, it is relatively easy to construct the relational schema. Each element node X that is a separate relation inlines all the nodes Y that are reachable from it such that the path from X to Y does not contain a node (other than X) that is to be made a separate relation. Inlining an element X into a relation corresponding to another element Y creates problems when an XML document is rooted at the element X. To facilitate queries on such elements we make use of isRoot fields. Figure 15 shows the schema derived from the DTD graph of Figure 11. One striking feature is the small number of relations compared to the *Basic* schema (Figure 13).

The element sharing in *Shared* has query processing implications. For example, a selection query over all authors accesses only one relation in *Shared* compared to five relations in *Basic*. Despite the fact that *Shared* addresses some of the shortcomings and shares some of the strengths of *Basic*, *Basic* performs better in one important respect – reducing the number of joins starting at a particular element node. In the next section, we explore a hybrid approach that combines the join reduction properties of *Basic* with the sharing features of *Shared*.

book (bookID: integer, book.booktitle.isroot: boolean, book.booktitle : string, author.name.firstname: string, author.name.lastname: string, author.address: string, author.authorid: string)
article (articleID: integer, article.contactauthor.isroot: boolean, article.contactauthor.authorid: string, article.title.isroot: boolean, article.title: string)
monograph (monographID: integer, monograph.parentID: integer, monograph.parentCODE: integer, monograph.title: string, monograph.editor.isroot: boolean, monograph.editor.name: string, author.name.firstname: string, author.name.lastname: string, author.address: string, author.authorid: string)
author (authorID: integer, author.parentID: integer, author.parentCODE: integer, author.name.isroot: boolean, author.name.firstname.isroot: boolean, author.name.firstname: string, author.name.lastname.isroot: boolean, author.name.lastname: string, author.address.isroot: boolean, author.address: string, author.authorid: string)

Figure 16: Relational Schema Generated using *Hybrid*

3.1.5. The Hybrid Inlining Technique

The Hybrid Inlining Technique, or *Hybrid*, is the same as *Shared* except that it inlines some elements that are not inlined in *Shared*. In particular, *Hybrid* additionally inlines elements with in-degree greater than one that are not recursive or reached through a "*" node. Set sub-elements and recursive elements are treated as in *Shared*. Figure 16 shows the relational schema generated using this hybrid approach. Note how this schema combines features of both *Basic* and *Shared* – *author* is inlined with *book* and *monograph* even though it is shared, while *monograph* and *editor* are represented exactly once.

So far, we have implicitly assumed that the data model is unordered, i.e., the position of an element does not matter. Order can, however, be easily incorporated into our framework by storing a position field for each element.

3.1.6. A Qualitative Evaluation of the Basic, Shared and Hybrid Techniques

In this section, we qualitatively evaluate our relation-conversion algorithms using 37 DTDs available from Robin Cover's XML pages [25]. We did not pose any criterion for selecting DTDs

except validity and availability for easy download. Some DTDs were excluded because they could not be parsed using the IBM alphaWorks xml4j XML parser [37].

3.1.6.1. Evaluation Metric

One of the major issues in evaluating the algorithms is the efficiency of query processing. Our metric is the *average number of SQL joins required to process path expressions of a certain length N*. We use this metric because (a) path expressions are at the heart of query languages proposed for semi-structured data, and (b) the evaluation of path expressions leads naturally to joins, and joins are typically the most expensive operation in a relational system.

This subsection logically contains "forward references" to Section 3.2, in which we describe how SQL queries are generated from semi-structured XML queries. However, the only point from Section 3.2 that is necessary to understand the results here is that a single semi-structured query could give rise to a union of several SQL queries, and that each of the queries may contain some number of joins. The use of *Basic* vs. *Shared* vs. *Hybrid* determines how many queries are generated, and how many joins are found in each query. Although *Basic* and *Hybrid* reduce the number of joins *per SQL query*, their higher degree of inlining could cause more SQL queries to be generated. For each algorithm, each DTD, and a variable number of path lengths, we made the following measurements:

1) The average number of SQL queries generated for path expressions of length N

2) The average number of joins in each SQL query for path expressions of length N

3) The total average number of joins in order to process path expressions of length N (the product of the previous two measurements)

In Sections 3.1.6.2 and 3.1.6.3, we assume that path expressions start from an arbitrary element in the DTD. We relax this assumption in Section 3.1.6.4.

3.1.6.2. Evaluation Results for Path Expressions of Length 3

In this section, we show the results for path expressions of length 3, which is the longest path length applicable to all 37 DTDs. We shall examine the results for other path lengths in the next section.

We first evaluate the practicality of the *Basic* approach. For 11 out of 37 DTDs, our implementation of the *Basic* approach did not run to completion because it ran out of virtual memory. The reason for this is that *Basic* generates a large number of tables if DTDs have large strongly connected components. We can see this effect clearly on some of the DTDs on which *Basic* did run to completion. One 19 node DTD has a strongly connected component of size 7, and the number of relations created is 204 times as many as created by *Hybrid*, totalling 3462 relations. This is because *Basic* creates a separate table for every possible path, which leads to the creation of an exponential number of tables in the presence of strongly connected components. Due to this severe limitation of *Basic*, we focus on the *Shared* and *Hybrid* approaches for the remainder of this evaluation.

Figure 17, Figure 18, and Figure 19 show our experimental results for 10 of the DTDs (the experimental results for the other DTDs are summarized later). As shown in Figure 17, *Hybrid*

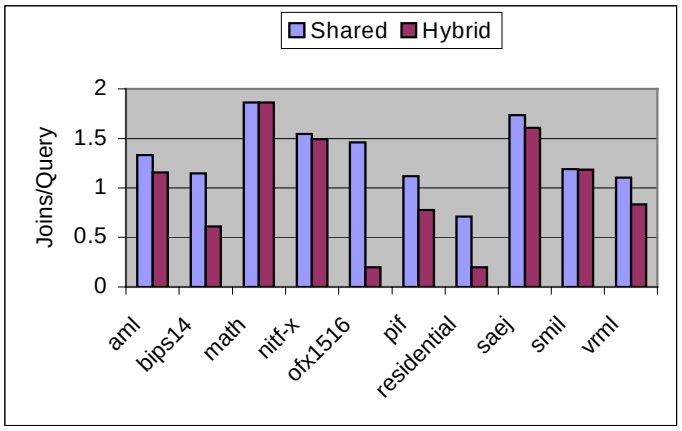

Figure 17: Average Number of Joins in each SQL Query for Path Expressions of Length 3

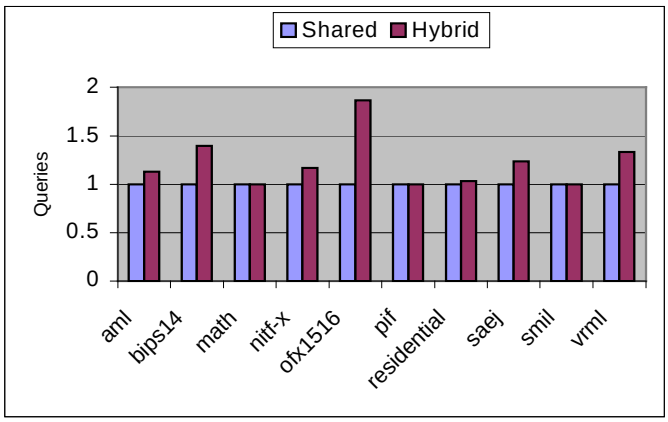

Figure 18: Average Number of SQL Queries for Path Expressions of Length 3

eliminates a large number of joins for some DTDs, whereas for others, *Hybrid* and *Shared* produce about the same number of joins. Figure 18 shows that for some DTDs, querying over 3-length path expressions using *Hybrid* requires more SQL queries than using *Shared*, while for other DTDs, the number of SQL queries is the same. Note that for any path expression, *Shared* always produces at least the number of joins per SQL query as *Hybrid*, and *Hybrid* always produces at least the number of SQL queries as *Shared*. Figure 19 shows the total number of joins.

Using the average total number of joins required to process path expressions of length 3, we can roughly categorize the 37 DTDs into four groups:

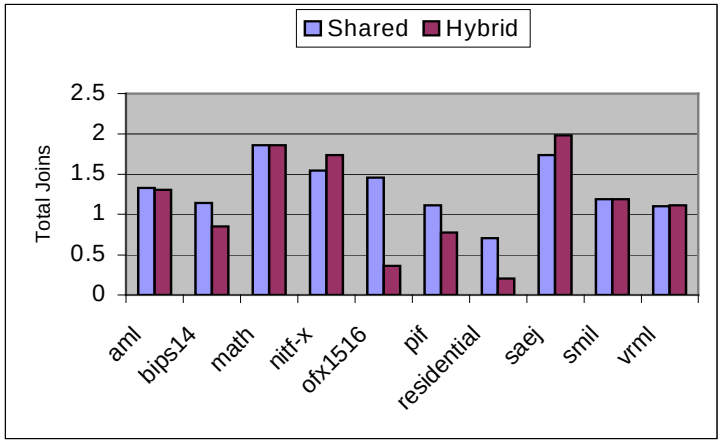

Figure 19: Total Average Number of Joins for Path Expressions of Length 3

Group 1: DTDs for which *Hybrid* reduces a large percentage of joins per SQL query but incurs a smaller increase in the number of SQL queries. The net result is that *Hybrid* requires fewer joins than *Shared*. An example of a DTD in this class is "ofx1516".

Group 2: DTDs for which *Hybrid* reduces a large percentage of joins per SQL query and incurs a comparable increase in the number of SQL queries. The total number of joins is about the same. An example of a DTD in this class is "vrml".

Group 3: DTDs for which *Hybrid* reduces some joins per SQL query, but not enough to offset the increase in the number of SQL queries. Therefore, *Hybrid* generates more joins for a path expressions than *Shared*. An example of a DTD in this class is "saej".

Group 4: DTDs for which both *Shared* and *Hybrid* produces about the same number of joins per SQL query, and about the same number of SQL queries, resulting in approximately the same total number of joins. An example of a DTD in this class is "math".

	Group 1	Group 2	Group 3	Group 4
Num DTDs	13	2	6	16

Table 1: Classification of DTDs into Groups

Hybrid inlines more than *Shared* in Groups 1, 2 and 3. This reduces the number of joins per SQL query but increases the number of SQL queries. The net increase or decrease in the total number of joins depends on the structure of the DTD. In Group 4, most of the shared nodes are either set nodes or involved in recursion. Since *Shared* and *Hybrid* treat set nodes and recursive nodes identically, there is no significant difference in their performance in Group 4.

The number of DTDs in each group from all 37 DTDs is summarized in Table 1. We can infer that in a large number of DTDs (Group 4), most of the shared nodes are either set nodes or recursive nodes.

3.1.6.3. Results for Path Expressions of Other Lengths

In the previous section, we showed the results for path expressions of length 3. In order to see how the results carry over to other path lengths, let us examine how the number of joins scales with the path length. We found that for all the DTDs, the number of joins scales linearly with the path length. The only difference is the scaling factor, which is determined by the structure of the DTD. Furthermore, the gap between the performance of *Shared* and *Hybrid* typically widens when the path lengthens. Figure 20 and Figure 21 show the scaling for two DTDs in group 1 and group 3 respectively.

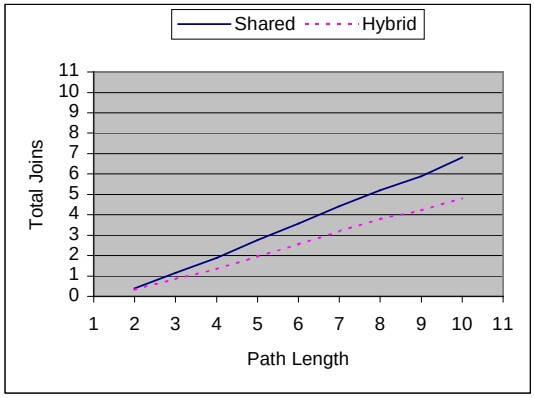

Figure 20: Scaling of Total Number of Joins with Path Length (Group 1 DTD)

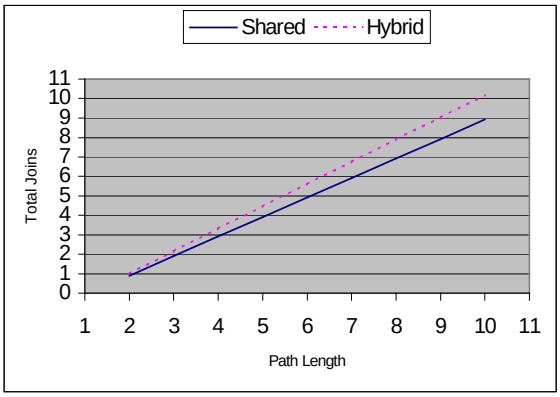

Figure 21: Scaling of Total Number of Joins with Path Length (Group 3 DTD)

3.1.6.4. Evaluation for Path Expressions Starting from the Document Root

So far, we have examined the performance of our algorithms assuming path expressions start from an arbitrary node in the DTD graph. What is difference if the path expressions start from the root of a document? The real difference is in the total number of joins. A path expression starting from the root of a document is always converted to one SQL query – therefore the total number of joins is equivalent to the number of joins per SQL query. Since the *Hybrid* algorithm always produces fewer joins per SQL query, it is always better than *Shared* for path expressions that start from the document root.

For DTDs in groups 3 and 4 (the majority of DTDs), both *Shared* and *Hybrid* are practically the same. The main issue is the excessive fragmentation of the DTDs that leads to the number of joins being almost equal to the length of the path expression (Figure 21). This is likely to be very inefficient in the relational model, especially for long path lengths. The main cause of this fragmentation is the presence of set sub-elements. Section 3.4 includes a proposed extension to alleviate this problem.

3.2. Converting Semi-structured Queries to SQL

Semi-structured query languages have a lot more flexibility than SQL. In particular, they allow path expressions with various operators and wild cards. The challenge is to rewrite these queries in SQL exploiting DTD information. In this section, we consider only queries with string values as results. Queries with more complex result formats are dealt with in Section 3.3. For ease of exposition, we present the translation algorithm only in the context of the *Shared* approach. The generalization to the other approaches is straightforward.

The remainder of this section is organized as follows. We first illustrate how queries having simple path expressions (having only the "." Operator) can be translated to SQL. We then generalize our translation to simple recursive path expressions (having only the "." and "*" operators). Finally, we show how existing techniques can be adapted to translate any regular path expression into (possibly many) simple recursive path expressions, which can then be translated to SQL.

```
WHERE <book>
   <booktitle> The Selfish Gene </booktitle>
   <author>
        <name>
            <firstname> $f </firstname>
            <lastname> $l </lastname>
        </name>
   </author>
          </book> IN * CONFORMING TO pubs.dtd
CONSTRUCT <result> $f $l </result>
```

```
Select Y.name.firstname,
       Y.name.lastname
From   book X, X.author Y
Where X.booktitle = "The Selfish Gene"
```

Figure 22: Example XML-QL and Lorel Queries with Simple Path Expressions

```
Select A."author.name.firstname",
       A."author.name.lastname"
From   author A, book B
Where B.bookID = A.parentID
       AND A.parentCODE = 0
       AND B."book.booktitle" = "The Selfish Gene"
```

Figure 23: SQL Query Generated for XML Query with Simple Path Expressions

3.2.1. Converting Queries with Simple Path Expressions to SQL

Consider the XML-QL query, and the equivalent Lorel query, shown in Figure 22. The query selects the first and last name of the author of the book with title "The Selfish Gene". Note that we have slightly extended the XML-QL syntax to query over all documents conforming to a DTD.

As can be seen from the Lorel-like representation, this query essentially consists of five path expressions, namely, *book*, *X.author*, *Y.name.firstname*, *Y.name.lastname* and *X.booktitle*. Of these path expressions, *book* is the root path expression and the others are dependent path expressions. This query is translated into SQL as follows. First, the relation(s) corresponding to start of the root path expression(s) are identified and added to the from clause of the SQL query. Then, if necessary, the path expressions are translated to joins among relations (when elements are inlined, joins are not necessary).

```
WHERE <*.monograph>
            <editor.(monograph.editor)*>
                <name> $n </name>
            </>
            <title> Subclass Cirripedia </title>
        </> IN * CONFORMING TO pubs.dtd
CONSTRUCT <result> $n </result>
```

```
Select Y.name
From   *.monograph X, X.editor.(monograph.editor)* Y
Where X.title = "Subclass Cirripedia"
```

Figure 24: Example XML-QL and Lorel Queries with Recursive Path Expressions

The SQL query generated in this fashion for the example query above is shown in Figure 23. Note that a join condition has been added to the "where" clause to link the book and author and a selection (A.parentCODE = 0, where 0 indicates that the parent of the author is a book) is performed on author to make sure that only authors reached through book are considered.

3.2.2. Converting Simple Recursive Path Expressions to SQL

Consider a query that selects the names of all editors reachable directly or indirectly from the monograph with title "Subclass Cirripedia". The corresponding XML-QL and Lorel queries are shown in Figure 24.

There are two interesting features about this query. The first is the tag "*.monograph" which states that we are interested in monographs reachable from any path. The second is the tag "editor.(monograph.editor)*" that specifies all editors reachable directly or indirectly from a monograph. The trick in converting this to a least fix-point query supported in SQL is to determine (a) the initialization of the recursion and (b) the actual recursive path expression. In the example above, the initialization of the recursion is the path expression *.monograph.editor with the selection condition monograph.title = "Subclass Cirripedia" and the recursive path expression is monograph.editor. Each can be converted to a SQL fragment just like a simple path expression. The final query is the union of the two SQL fragments within a least fix-point operator. The query

```
With Q1 (monographID, name) AS
(Select X.monographID, X."editor.name"
 From monograph X
 Where X.title = "Subclass Cirripedia"
UNION ALL
 Select Z.monographID, Z."editor.name"
 From Q1 Y, monograph Z
 Where Y.monographID = Z.parentID AND
        Z.parentCODE = 0
)
Select A.name
From Q1 A
```

Figure 25: SQL Query Generated from Recursive Path Expressions

```
Select X
From monograph.(#)*.name X
```

Figure 26: Example XML-QL Query with a Complex Recursive Path Expression

generated in this fashion is shown in Figure 25, in SQL'92 [7] syntax. Note that the "with clause" is used to specify the least fix-point operator in SQL.

3.2.3. Converting Arbitrary Path Expressions to Simple Recursive Path Expressions

In general, path expressions can be of arbitrary complexity. For example, we could have a query that asks for all the name elements reachable directly or indirectly through *monograph*. This would be represented in a Lorel-like language as shown in Figure 26 (an equivalent query can be expressed in XML-QL).

Our approach is to take path expressions that appear in such queries (in this example "monograph.(#)*.name") and translate them into possibly many simple (recursive) path expressions. In order to do this, we rely on existing techniques, developed in the context of compile-time optimization of semi-structured queries [30][50], for converting arbitrary path

expressions to simple path expressions exploiting schema information. SQL queries are then generated for each simple recursive path expression. This notion of splitting a path expression to many simple path expressions is crucial to processing queries having arbitrary path expressions in SQL.

In our example, given the DTD of the XML documents, the path expression "monograph.(#)*.name" would be expanded to the simple (recursive) path expressions "monograph.(editor.monograph)*.author.name" and "monograph.(editor.monograph)*.editor.name". Each of these can then be converted to SQL using the least fix-point operator (as described above), and their results can be unioned together to produce the desired output.

The above technique is general enough to produce simple recursive path expressions with nested recursion (e.g., "(a.(b)*.c)*"). However, SQL does not currently have support for nested fix-point operators and hence, these queries cannot be converted to SQL.

3.3. Converting Relational Results to XML

In the previous section, we assumed that the results of a query were string values. We relax this assumption in this section and explore how the tabular results returned by SQL queries can be converted to complex structured XML documents. This is perhaps the main drawback in using current relational technology to provide XML querying – constructing arbitrary XML result sets is difficult. In this section we give some examples, using XML-QL as the illustrative query language because it provides XML structuring constructs. We will also return to this issue in Chapter 4.

```
WHERE <book>
            <author>
                <firstname> $f </firstname>
                <lastname> $l </lastname>
            </author>
        </book> IN * CONFORMS TO pubs.dtd
CONSTRUCT  <author>
                    <firstname> $f </firstname>
                    <lastname> $l </lastname>
                </author>
```

Figure 27: An XML-QL Query Performing Simple Structuring

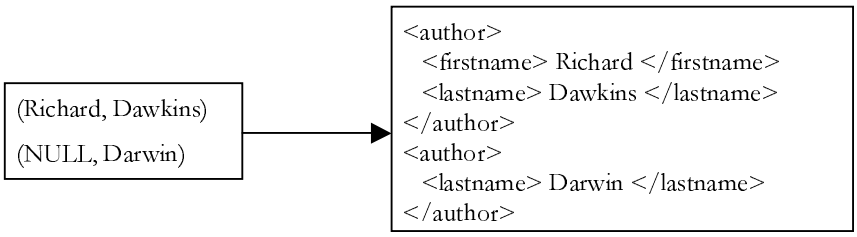

Figure 28: Producing Simple Structured XML Output

3.3.1. Simple Structuring

Consider the query in Figure 27 that asks for the first name and last names of all the authors of books, nested appropriately. Constructing such results from a relational system is natural and efficient, since it only requires attaching the appropriate tags for each tuple (Figure 28).

3.3.2. Tag Variables

A tag variable is one that ranges over the value of an XML tag. Some queries requiring tag variables in their results are naturally translated to the relational model. Consider the query in Figure 29, which asks for names of authors of all publications, nested under a tag specifying the type of publication. This can be handled by generating a relational query that contains the tag value as an element of the result tuple. Then at result generation time, the tag attribute in the result tuple can be converted to the appropriate XML tag (Figure 30).

```
WHERE <$p>
          <author>
             <firstname> $f </firstname>
             <lastname> $l </lastname>
          </author>
        </> IN * CONFORMS TO pubs.dtd
CONSTRUCT <$p>
                  <author>
                     <firstname> $f </firstname>
                     <lastname> $l </lastname>
                  </author>
               </>
```

Figure 29: An XML-QL Query with Tag Variables

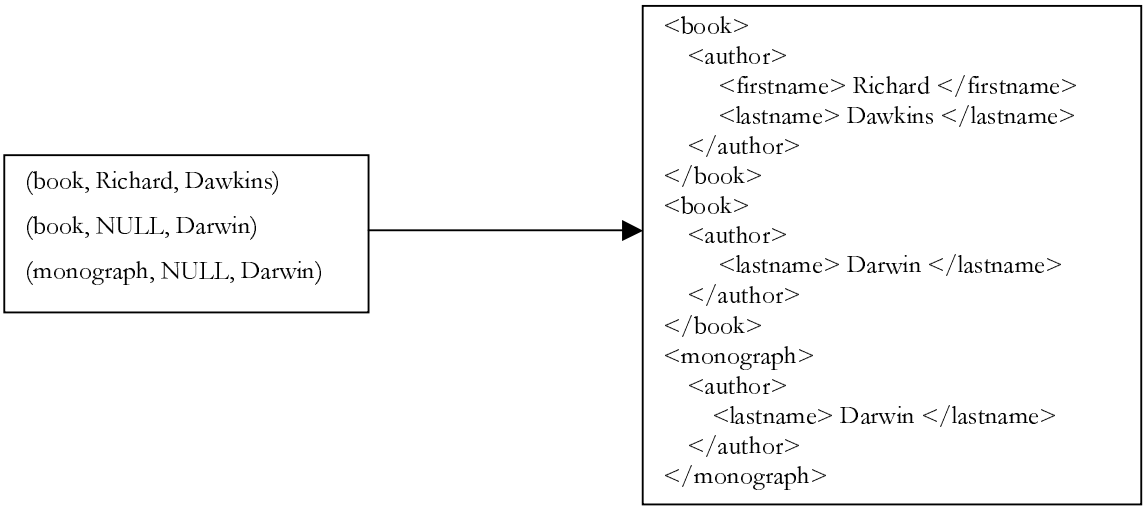

Figure 30: Producing XML Output with Tag Variables

3.3.3. Grouping

Consider the query in Figure 31 that requires all the publications of an author (assuming an author is uniquely identified by his/her last name) to be grouped together, and within this structure, requires the titles of publications to be grouped by the type of the publication. The relational result from the translation of this query will be a set of tuples having fields corresponding to last name of author, title of publication and type of publication. However, we cannot use the relational group-by operator to group by last name and type of publication because the SQL group-by semantics

```
WHERE <$p>
          <(title|booktitle)> $t </>
          <author>
              <lastname> $l </lastname>
          </author>
       </> IN * CONFORMS TO pubs.dtd
CONSTRUCT <author ID=authorID($l)>
                  <name> $l </name>
                  <$p ID=pID($p)>
                      <title> $t </>
                  </>
              </author>
```

Figure 31: An XML-QL Query Performing Grouping

implies that we must apply an aggregate function to title, which does not make sense. Thus, the options are either (a) have the relational engine order the result tuples first by last name and then by type and scan the result in order to construct the XML document or (b) get an unordered set of tuples and do a grouping operation, by last name and then by type, outside the relational engine. The first approach is illustrated in Figure 32.

Figure 32 illustrates two points. The first is that treating tag variables as attributes in the result relation provides a way of uniformly treating the contents of the result XML document. In this case, we are able to group by the tag variable just like any other attribute. The second observation is that some relational database functionality (hash-based group-by) is either not fully exploited or is duplicated outside.

3.3.4. Complex Element Construction

Unfortunately, returning tag values as tuple attributes cannot handle all result construction problems. In particular, queries that are required to return complex XML elements are problematic. As an example, consider the query in Figure 33 that asks for all article elements in the XML data set, and furthermore assume that an article may have multiple authors and multiple

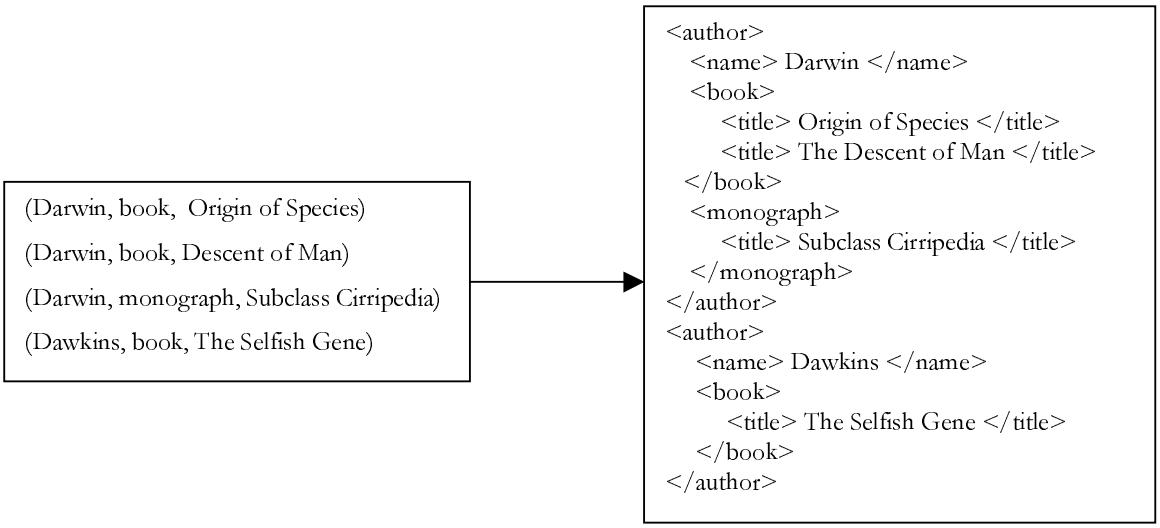

Figure 32: Producing Grouped XML Output

```
WHERE <article>
            <$p> $y </>
        </article> IN * CONFORMING TO pubs.dtd
CONSTRUCT <$p> $y </>
```

Figure 33: An XML-QL Query Constructing a Complex XML Element Result

titles. In object-relational terminology, article has two set-valued attributes, authors and titles, corresponding to two set sub-elements in XML terminology.

To create the appropriate result, we must retrieve all authors and all titles for each article. This is difficult to do efficiently in the relational model because flattening multiple set-valued attributes into tuple format gives rise to a multi-valued dependency [29] and is likely to be very inefficient when the sets are large, for example, if papers have many authors and many titles. Another approach would be to return separate relations, each flattening one set-valued attribute, and "join" these relations outside the database while constructing the XML document. However, this requires duplication of database functionality both in terms of execution and optimization. This solution would be particularly bad for an element with many set-valued attributes. A related problem occurs

```
WHERE <book>
            <article> $a </article>
        </book> IN * CONFORMS TO pubs.dtd
CONSTRUCT <article> $a </>
```

Figure 34: An XML-QL Query Producing Heterogeneous XML Output

when reconstructing recursive elements. We return to this issue of constructing complex XML elements in Section 3.4 and Chapter 4.

3.3.5. Heterogeneous Results

Consider the XML-QL query in Figure 34, which creates a result document having both titles and authors as elements (this is the heterogeneous result). This is easily handled in our approach for translating queries because this query would be split into two queries, one for selecting titles and another for selecting authors. The results of the two queries can be handled in different ways, one constructing title elements and another constructing author elements. The results can then be merged together.

3.3.6. Nested Queries

XML query languages such as XML-QL and XQuery are structured in terms of query blocks, where one query block can be nested under another. These nested queries can be rewritten in terms of SQL queries, using outer joins to construct the association between a query and a sub-query. More details regarding this are presented in Chapter 4.

3.4. Proposed Extensions to Relational Systems

Our experience with the implementation of the above techniques has shown that relational systems can handle XML query workloads more effectively with the following extensions:

Support for Sets: Set-valued attributes would be useful in two important ways. First, storing set sub-elements as set-valued attributes [81] would reduce fragmentation. This is likely to be a big win because most of the fragmentation we observed in real DTDs was due to sets. Second, set-valued attributes, along with support for nesting [39], would allow a relational system to perform more of the processing required for generating complex XML results.

Untyped/Variable-Typed References: IDREFs are not typed in XML. Therefore, queries that navigate through IDREFs cannot be handled in current relational systems without a proliferation of joins – one for each possible reference type.

Information Retrieval Style Indices: More powerful indices, such as Oracle's ConText search engine for XML [53], that can index over the structure of string attributes would be useful in querying over ANY fields in a DTD. Further, under restricted query requirements, whole fragments of a document can be stored as an indexed text field, thus reducing fragmentation.

Flexible Comparisons Operators: A DTD schema treats every value as a string. This often creates the need to compare a string attribute with, say, an integer value, after typecasting the string to an integer. The traditional relational model cannot support such comparisons. The problem persists even if we use XML schemas (which have more sophisticated types) because different schemas may represent "comparable" values as different types. A related issue is that of flexible indices. Techniques for building such indices have been proposed in the context of semi-structured databases [49].

Multiple-Query Optimization/Execution: As outlined in Section 3.2.3, complex path expressions are handled in a relational database by converting them into many simple path expressions, each corresponding to a separate SQL query. Since these SQL queries are derived from a single regular path expression, they are likely to share many relational scans, selections and joins. Rather than treating them all as separate queries, it may be more efficient to optimize and execute them as a group [63].

More Powerful Recursion: As mentioned in Section 3.2.3, in order to fully support all recursive path expressions, support for fixed point expressions defined in terms of other fixed point expressions (i.e., nested fixed point expressions) is required.

These extensions are not by themselves new and have been proposed in other contexts. However, they gain new importance in light of our evaluation of the requirements for processing XML documents. Further research on these techniques in the context of processing XML documents will, we believe, facilitate the use of sophisticated relational data management techniques in handling the novel requirements of emerging XML-based applications.

CHAPTER 4: EFFICIENTLY MATERIALIZING RELATIONAL DATA AS XML DOCUMENTS

Most operational business data, even for new web-based applications, is stored in relational database systems. This is unlikely to change in the foreseeable future because of the reliability, scalability, performance and tools associated with relational database systems. Consequently, if XML is to fulfill its potential as the Internet data exchange format, some mechanism is needed to materialize relational data in the form of XML documents. This chapter addresses the issue of efficiently materializing relational data as XML documents.

There are two main requirements for efficiently materializing 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 the contributions of this chapter 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 remainder of this chapter is organized as follows. In Section 4.1 we present our SQL-based language specification for specifying the conversion from relational data to XML documents. In Section 4.2 we explore a range of implementation alternatives and in Section 0 we evaluate the performance of the alternatives and show the superiority of the "outer union" plans. In Section 4.4 we outline the algorithm to generate the "outer union" plans from the SQL query specification proposed in this chapter.

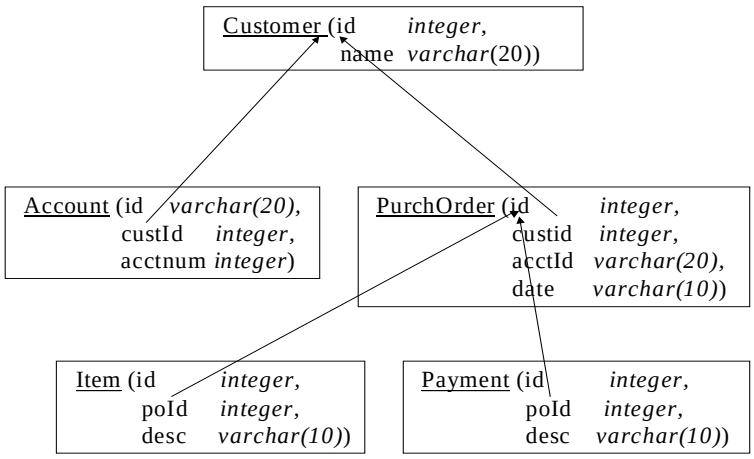

Figure 35: Customer Relational Schema

4.1. 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 language 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.

As an example, consider the relational schema shown in Figure 35. 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. Let us now suppose that we wish to covert the data in this relational schema into the XML document in Figure 36. As shown in the XML document, each customer element has an id attribute and a name sub-element. Each customer element also has its associated accounts and purchase orders nested under it. The purchase order elements associated with a customer are ordered by their date. Each purchase order element in turn has its associated items and payments nested under it.

```
<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 36: An XML Document Describing a Customer

To convert the data in the relational schema in Figure 35 into the XML document in Figure 36, we can write a SQL query that follows the nested structure of the document, as shown in Figure 37. The query 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

```
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 37: 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 38: Definition of an XML Constructor

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 38. 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. 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 37 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. In our proposed 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 37, where the purchase orders of a customer are ordered by their date before being aggregated by the XMLAGG function.

4.2. 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 chapter 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 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

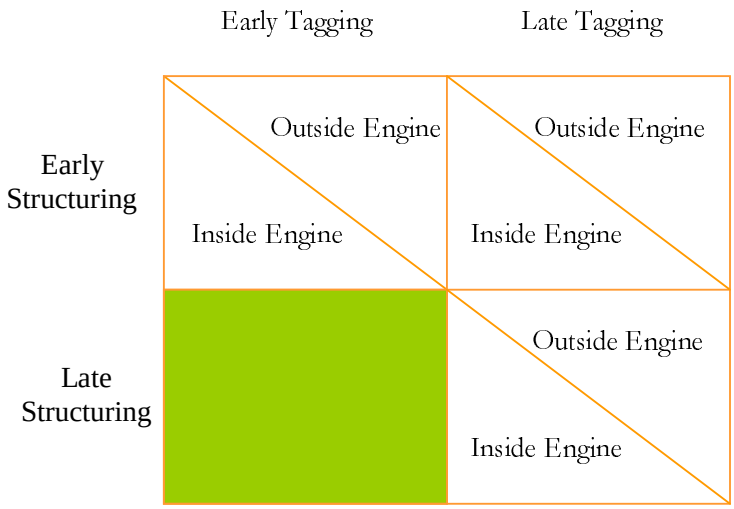

Figure 39: Space of Alternatives for Publishing XML

done earlier (*early structuring*). These two dimensions of tagging and structuring give rise to a space of alternatives shown pictorially in Figure 39.

Each alternative in this space has variants depending on how much work is done inside the relational engine. "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. 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.

4.2.1. 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.

4.2.1.1. 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 36. 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 4.3. 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.

4.2.1.2. 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 4.1. 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 37 by executing correlated sub-queries for nested queries. This is depicted pictorially in Figure 40. 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.

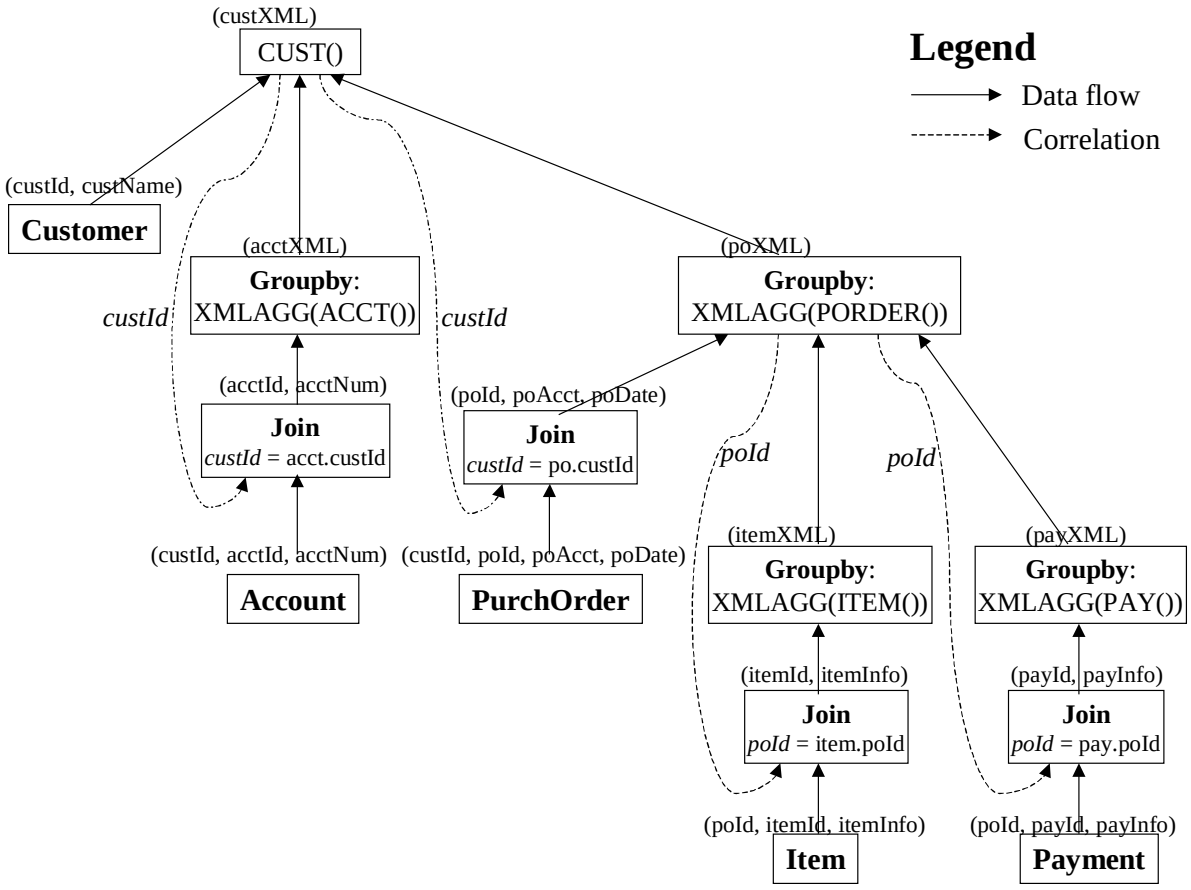

Figure 40: SQL Query Execution Plan for the Correlated CLOB Approach

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.

4.2.1.3. 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 [64] inside the relational engine to give the relational optimizer more flexibility. The query execution plan obtained by de-correlating the query in Figure 37 is shown in Figure 41.

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 41. 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 41. 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 41).

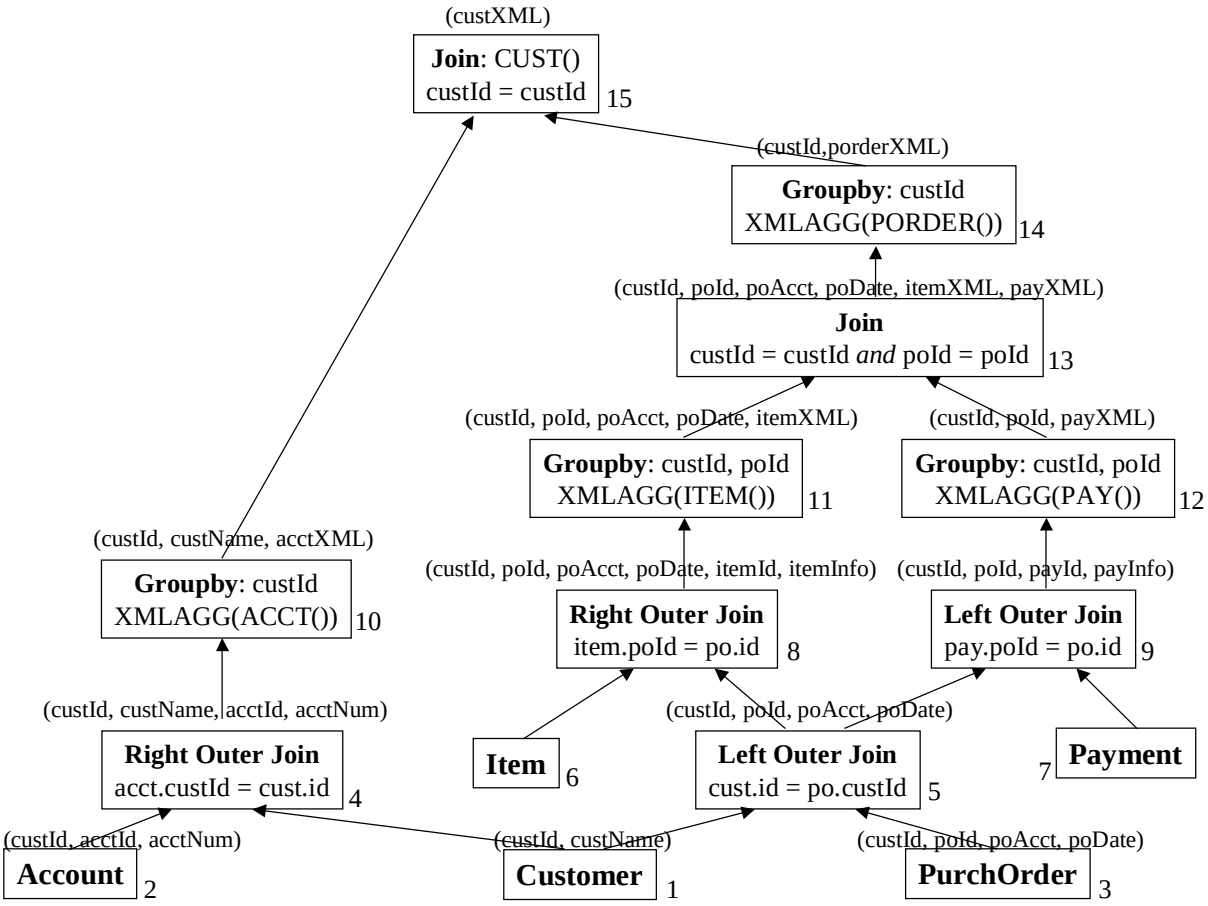

Figure 41: SQL Query Execution Plan for the Decorrelated CLOB Approach

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.

4.2.2. 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

```
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 42: SQL Query for the Redundant Relation Approach

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.

4.2.2.1. 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 42.

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 42 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 [29] 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 affect performance.

4.2.2.2. 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 43 shows a query execution plan that reduces content redundancy for the query of Figure 37. The SQL query corresponding to this query execution plan is also shown in Figure 44. 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 43, and correspond to lines 5-8, 13-17 and 18-21 respectively in the SQL query of Figure 44 (these are defined as the inline views custAcct, custPorderItem and

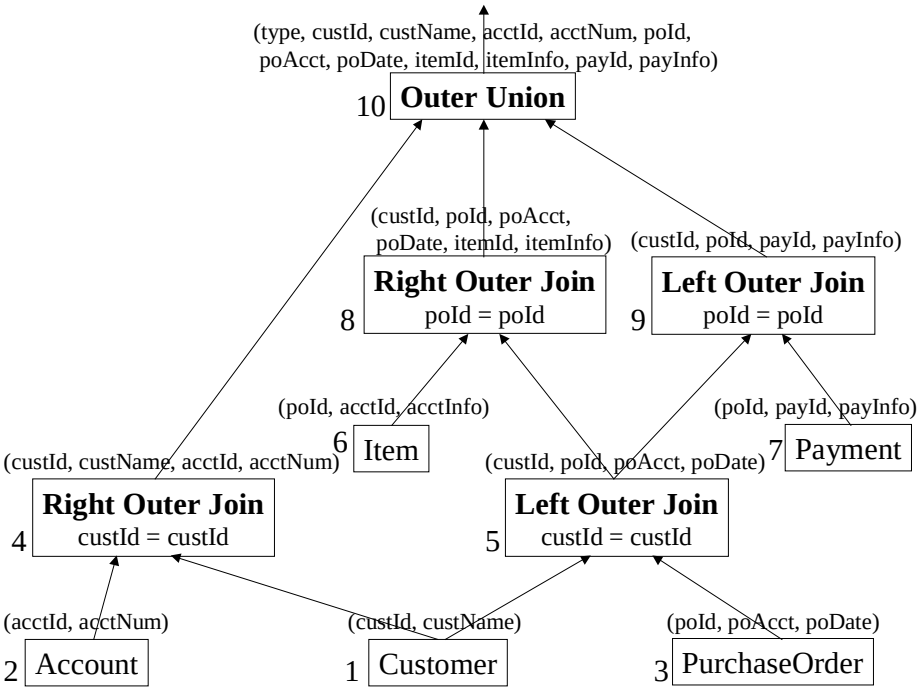

Figure 43: SQL Query Execution Plan for the (Unsorted) Path Outer Union Approach

custPorderPay in the SQL query using the "with" statement [7][18]). As in the Decorrelated 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 43). 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

```
-- 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 44: SQL Query for the (Unsorted) Path Outer Union Approach

is to union the contents of all leaf level elements. 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 deal with 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 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 43 and corresponds to lines 23-30 in Figure 44.

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 the *outer union* of each *path* from the root-level table to a leaf-level table.

4.2.2.3. 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 45 shows the query execution plan for the Node Outer Union approach for our running example. Figure 46 shows the corresponding SQL query. Note how all the parent 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

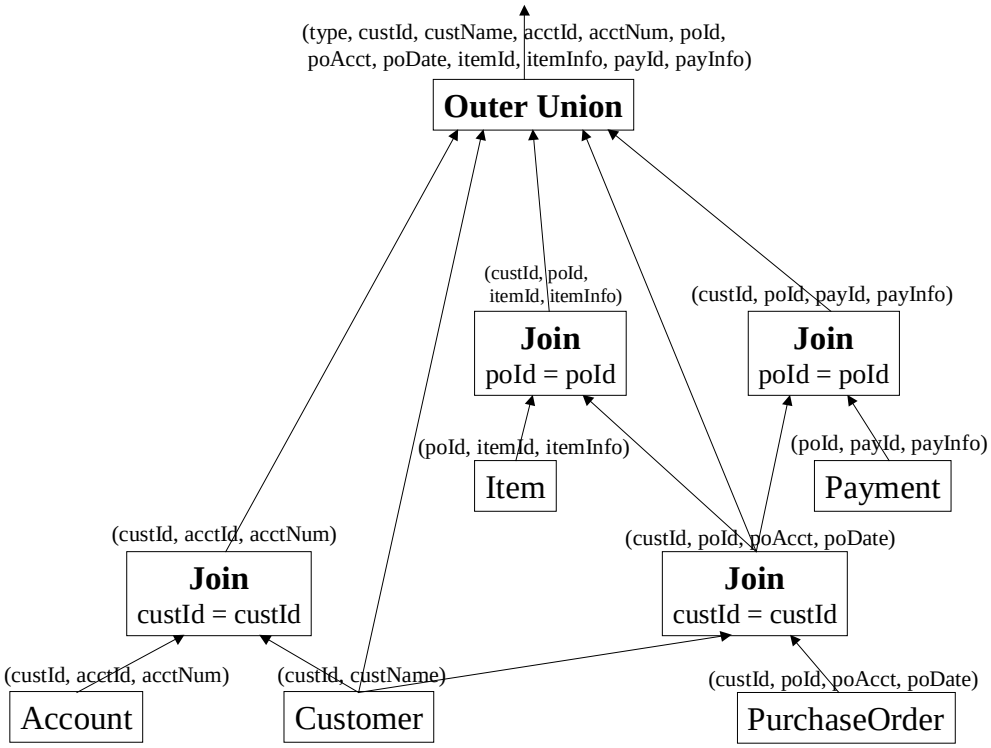

Figure 45: SQL Query Execution Plan for the (Unsorted) Node Outer Union Approach

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. Although only a subset of the columns in a given tuple will have data values, in the

```
-- 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 46: SQL Query for the (Unsorted) Node Outer Union Approach

absence of null-value compression, this may lead to increased processing overhead due to larger tuple widths.

4.2.2.4. 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 is invoked as the last processing step, after the relational content has been produced. This (single) aggregate function performs 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 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 [70]. Exactly how to do this partitioning (and merging) is left for future work.

4.2.3. 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.

4.2.3.1. 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.

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 47 and Figure 48 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 47. 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 this 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 ensure that tuples with null values in their sort columns occur first, parent tuples (customers) will always occur before child tuples

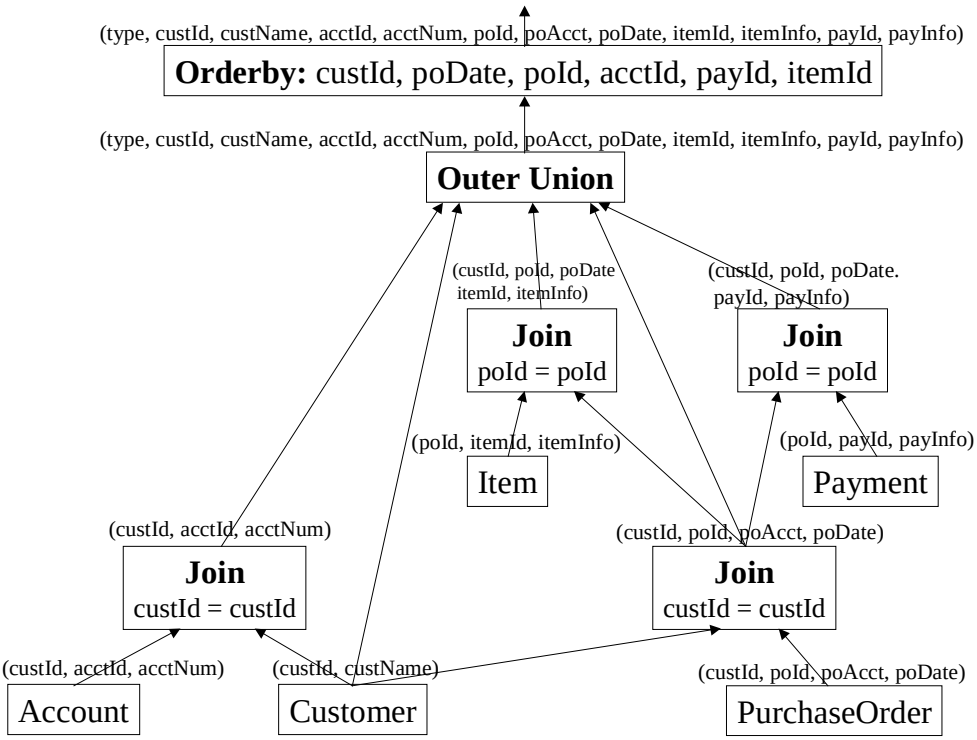

Figure 47: SQL Query Execution Plan for the Sorted Node Outer Union Approach

(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

```
-- Lines 1-37 compute the node outer union as in Figure 46
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 48: SQL Query for the Sorted Node Outer Union Approach

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 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.

4.2.3.2. 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

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.

Table 2: Summary of Approaches for Publishing Relational Data as XML Documents

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.

4.3. 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 Table 2. 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). 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.

4.3.1. 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 37 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 37 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 37, 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 37 is not balanced because it satisfies condition (a) but not condition (b). 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 38). 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 37 and Figure 35) 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

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.

Table 3: Experimental Parameters

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 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 Table 3.

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.

4.3.2. Experimental Setup

To conduct our performance comparison, we implemented the various alternatives discussed in Section 4.2 in the code base of the DB2 Universal Database system [18]. 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 400 MHz Pentium II processor with 256 MB of main memory running Windows NT 4.0.

For the experiments, we varied the parameters discussed in Section 4.3.1 in the ways shown in Table 4. 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

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

Table 4: Parameter Settings for Experiments

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 4.3.7, 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 4.3.8.

4.3.3. 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 49 and Figure 50, 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 throughout all of our experiments, so we will not examine it further.

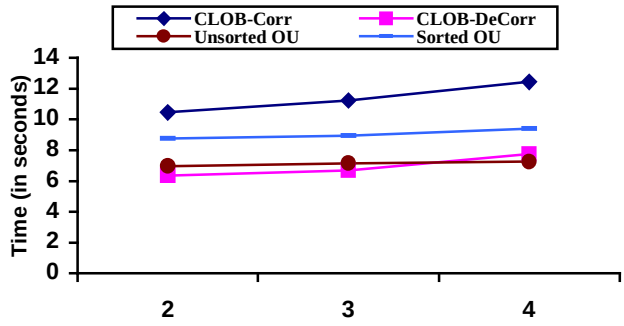

Figure 49: Varying Query Fan Out (Inside the Engine)

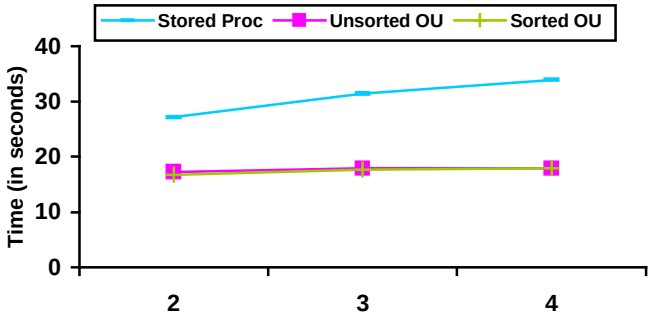

Figure 50: Varying Query Fan Out (Outside the Engine)

The interesting thing to note in Figure 49 and Figure 50 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.

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

Figure 51: Break Down of XML Construction Time

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 51 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 to 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.

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 chapter.

4.3.4. Effect of Query Fan Out

We now return to Figure 49 and Figure 50 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 49. 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 50 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.

4.3.5. Effect of Query Depth

We now turn our attention to the next parameter – query depth. Figure 52 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

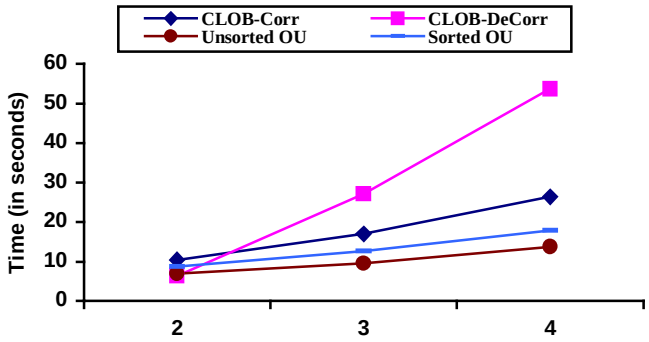

Figure 52: Varying Query Depth (Inside the Engine)

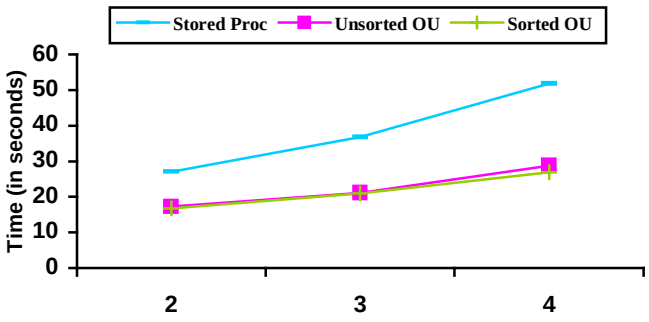

Figure 53: Varying Query Depth (Outside the Engine)

temporary space and materialized again later. This problem 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 53.

4.3.6. 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

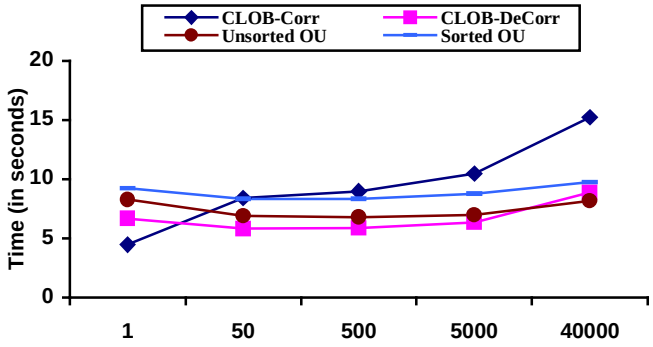

Figure 54: Varying Number of Roots (Inside the Engine)

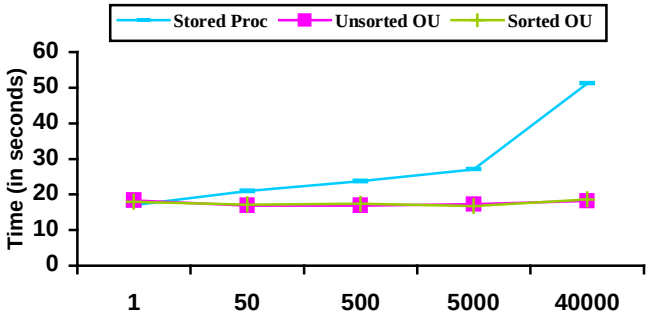

Figure 55: Varying Number of Roots (Outside the Engine)

"inside the engine" approaches (see Figure 54). 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 55). The relative performance of the outer union approaches remains unchanged.

4.3.7. 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.

4.3.8. 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.

4.3.9. 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 Table 2) – 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.

4.4. 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 chapter 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 chapter) 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 chapter, the SQL query rewrite module [56] 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 chapter, the algorithms should generalize to other query languages that use nested sub-queries to create nested document structures. Examples of such query languages include XQuery [79], XML-QL [27] and RXL [31].

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.

4.4.1. Input Parameters for Outer Union SQL Generation

Using the language extensions proposed in this chapter, a SQL query specifying the construction of XML documents can be represented as the template shown in Figure 56. 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

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

Figure 56: Template of Top-level SQL Query Specifying XML Construction

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

Figure 57: Template of SQL Sub-Query Specifying XML Construction

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.

As an example, consider the SQL query shown in Figure 37. 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 57. 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 37 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.

4.4.2. Generating SQL Queries for the Path Outer Union Approaches

As described in Sections 4.2.2.2 and 4.2.3.1, 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.

Figure 58 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.

```
01. Algorithm buildPaths (Query 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.
12.      // Propagate 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 58: Algorithm to Generate Paths for the Path Outer Union Approaches

We will now walk through the algorithm using the example query shown in Figure 37 and illustrate how the SQL paths (lines 1-21) are generated for the path outer union query shown in Figure 44. The algorithm buildPath is first invoked with the *sqlQuery* parameter set to be equal to the top-level SQL query in Figure 37. 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 58) and this generates the cust inline view in the SQL query of Figure 44 (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, which 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 58). 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 58). 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 44). Further recursive invocations of the algorithm produce the other inline views, namely custPorderItem and custPorderPay.

```
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 59: Algorithm to Generate the Outer Union for the Path Outer Union Approaches

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 44). The high-level pseudo-code for generating the SQL for the outer union is given in Figure 59. 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 59). 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 60. 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.

```
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 60: Algorithm to Generate SQL for the Unsorted Path Outer Union Approach

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.

4.4.3. Generating SQL Queries for the Node Outer Union Approaches

As described in Sections 4.2.2.2 and 4.2.2.3, 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 44 and Figure 46) and hence requires different SQL generation algorithms. In this section, we thus present algorithms for generating SQL queries for the node outer union approaches.

```
01. Algorithm buildPaths (Query 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.
12.      // Propagate 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 61: Algorithm to Generate Paths 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 46 and Figure 48). The algorithm to generate the desired paths if given in Figure 61. This algorithm is broadly similar to the corresponding algorithm for the path outer union approaches (see Figure 58). There are, however, two important differences. First, the algorithm for the node outer union approaches does not have the logic to propagate 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

```
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 62: Algorithm to Generate SQL for the Unsorted Node Outer Union Approach

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 61.

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 46). This algorithm (not shown) is very similar to the corresponding algorithm for the unsorted path outer union approach (Figure 59), 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 62.

We now turn our attention to generating SQL for the sorted node outer union approach. As described in Section 4.2.3.1, 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 63 shows the algorithm for generating the SQL query for the sorted node outer union approach. The first few steps (lines 2-8) essentially produce the SQL for the unsorted

```
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 63: Algorithm to Generate SQL for the Sorted Node Outer Union Approach

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 48.

The next part of the SQL generation algorithm (lines 10-22 in Figure 63) 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 48. In order to create the right ordering sequence, which satisfies the conditions outlined in Section 4.2.3.1, 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.

CHAPTER 5: QUERYING XML VIEWS OF RELATIONAL DATA

The previous chapter considered the problem of efficiently publishing relational data as XML documents. This can be equivalently viewed as efficiently materializing XML views of relational data. However, in many cases, applications do not require all the data in an XML view to be materialized. For example, in an XML view of available items, an application developer may only be interested in a particular item at a given point in time. Materializing the availability of all the items would be wasteful in this case because it would result in unnecessary computation. A better solution is to support queries over XML views so that only the data items of interest are retrieved. Supporting queries over XML views also allows application developers to synthesize data from different XML views.

In this chapter, we consider the problem of evaluating XML queries over XML views of relational data. We consider the case where views and queries are specified using XQuery [79], the XML query language currently being standardized by the World Wide Web Consortium. We focus on two issues. The first is the design of a general framework for processing arbitrarily complex XQuery queries, including queries with features such as nested expressions and nested order. The other area of focus is performance, whereby we present techniques for efficiently evaluating XQuery queries over XML views of relational data. One such technique is XML view composition, which eliminates the construction of all intermediate XML fragments that do not appear in the final query result. Another performance-enhancing technique is what we call

"computation push-down". This pushes all data- and memory-intensive computation in an XQuery query down to the relational engine. As a result, the query processing power of a relational engine is used to efficiently evaluate XML queries. Only a small memory-efficient tagger is required outside the relational engine to tag the SQL results and produce the resulting XML.

We have implemented the techniques described above in the context of the XPERANTO middleware system [17], which works on top of any relational database system. During the course of our implementation, we have identified some of the limitations that arise from using a relational query processor for executing XML queries. These limitations are due to the semantic mismatch between XQuery and SQL. We identify the causes for this mismatch and propose possible solutions to help overcome this problem.

To summarize, the contributions of this chapter are: (a) a general framework for processing XQuery queries with features such as nested expressions and order, (b) a view composition mechanism that eliminates the construction of all intermediate XML fragments, (c) a computation push down mechanism that pushes all data and memory intensive computation in an XQuery query down to SQL, (d) a description of extensions that can enable a relational engine to handle a larger class of XQuery queries.

The remainder of this chapter is organized as follows. Section 5.1 describes our high-level query processing architecture. The next three sections describe the query processing components such as the parser, the view composition module and the computation pushdown module. Section 5.5 discusses implementation and performance aspects. Section 5.6 identifies limitations of the described approach, and proposes possible solutions.

5.1. Query Processing Architecture

In this section, we present our high-level query-processing architecture. We begin by illustrating how XML views are created and queried in XPERANTO.

As a starting point, XPERANTO automatically creates a *default XML view,* which is a low-level XML view of the underlying relational database. Users can then define their own views on top of the default view using XQuery. Moreover, views can be defined on top of views to achieve higher levels of abstraction. The main advantage of this approach is that a standard XML query language is used to create and query views. This is in contrast to approaches such as [19][31][51], where a proprietary language is used to define the initial XML view of the underlying relational database.

As an example of a default XML view, consider the simple purchase-order database shown in Figure 64. As shown, the database consists of three tables, one table to keep track of customer orders, a second table to keep track of the items associated with an order, and a third table to keep track of the payments due for each order. Items and payments are related to orders by an order id (oid). The default XML view corresponding to this database is shown in Figure 65. In the default XML view, top-level elements correspond to tables with table names appearing as tags (there is no specific ordering among the table elements). Row elements are nested under the table elements. Within a row element, column names appear as tags and column values appear as text.

Continuing the example, suppose a user wants to publish the purchase-order database as a list of orders in the XML format shown in Figure 66. There, each order appears as a top-level element, with its associated items and payments (ordered by due date) nested under it. To transform the

order

id	custname	custnum
10	Smith Construction	7734
9	Western Builders	7725

item

oid	desc	cost
10	generator	8000
10	backhoe	24000

payment

oid	due	amt
10	1/10/01	20000
10	6/10/01	12000

Figure 64: An Example Purchase-Order Database

```
<db>
  <order>
    <row> <id>10 </id> <custname> Smith Construction </custname> <custnum> 7734 </custnum> </row>
    <row> <id> 9 </id> <custname>Western Builders </custname> <custnum> 7725 </custnum> </row>
  </order>
  <item>
    <row> <oid> 10 </oid> <desc> generator </desc> <cost> 8000 </cost> </row>
    <row> <oid> 10 </oid> <desc> backhoe </desc> <cost> 24000 </cost> </row>
  </item>
  <payment>
    … similar to <order> and <item>
  </payment>
</db>
```

Figure 65: The Default XML View for the Purchase-Order Database

default view into the desired XML format, a user-defined view called "orders" is created, as shown in Figure 67.

As shown in the figure, an XQuery FLWR expression (short for For-Let-Where-Return expression) in lines 2-23 is used to construct each order element. The "for" clause on line 2 causes the variable $order to be bound to each "row" element of the order table. The XPath [76] expression appearing in line 2 describes how to extract each "row" element from the order table. It basically says to start at the root of the default view, navigate to each "order" element nested under it, and then navigate to each "row" element nested under those "order" elements. The constructor for each new "order" element is given in lines 4-23. For a given order, nested FLWR expressions are used to construct its list of associated items (lines 7-12) and payments (lines 15-21). The predicate on line 8 ($order/id = $item/oid) is used to join an order with its items. Similarly, the predicate on line 16 ($order/id = $payment/oid) is used to join an order with its payments. The payments are ordered by their due dates (line 21).

```
<order id="10">
  <customer> Smith Construction </customer>
  <items>
    <item description="generator" >
       <cost> 8000 </cost>
    </item>
    <item description="backhoe">
       <cost> 24000 </cost> </item>
    </item>
  </items>
  <payments>
    <payment due="1/10/01">
       <amount> 20000 </amount>
    </payment>
    <payment due="6/10/01">
       <amount> 12000 </amount>
    </payment>
  </payments>
</order>
<order id="9" >
  …
</order>
```

Figure 66: XML Purchase Orders

```
01. create view orders as (
02.   for $order in view("default")/order/row
03.   return
04.     <order id=$order/id>
05.       <customer> $order/custname </customer>
06.       <items>
07.         for $item in view("default")/item/row
08.         where $order/id = $item/oid
09.         return
10.           <item description=$item/desc >
11.             <cost> $item/cost </cost>
12.           </item>
13.       </items>
14.       <payments>
15.         for $payment in view("default")/item/row
16.         where $order/id = $payment/oid
17.         return
18.           <payment due=$payment/date>
19.             <amount> $payment/amount </amount>
20.           </payment>
21.         sortby(@due)
22.       </payments>
23.     </order>)
```

Figure 67: User-defined XML View

```
1. for $order in view("orders")
2. where $order/customer/text() like "Smith%"
3. return $order
```

Figure 68: Query over User-defined XML View

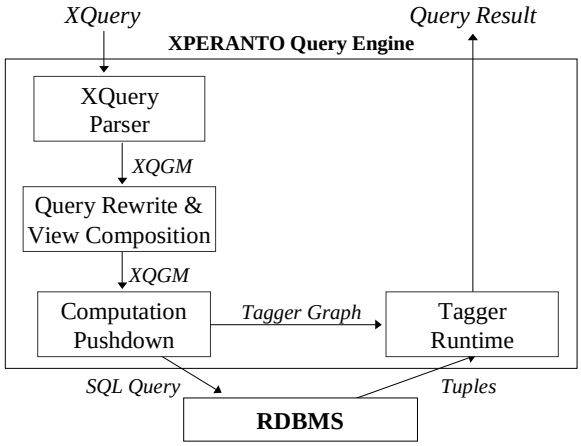

Figure 69: Query Processing Architecture

Once the "orders" view has been created, queries can be issued against it. This is illustrated in Figure 68, which shows a query that extracts all orders for the customer whose name begins with "Smith".

We now describe our query-processing architecture, which is shown in Figure 69. When an XQuery query is issued over XML views, it is first parsed and converted to an internal query representation called XML Query Graph Model (XQGM). The query is then composed with the views it references and rewrite optimizations are performed to eliminate the construction of intermediate XML fragments and push down predicates. The modified XQGM is then processed by the Computation Pushdown module, which separates the XQGM into two parts. The first part captures all the memory and data intensive processing and is pushed down to the relational database engine as a single SQL query. The second part is a tagger graph structure, which the

Tagger Runtime module uses to construct the XML query result. This is done in a single pass over the row streams resulting from the SQL query. In the remainder of this chapter, we describe the major query processing components in detail.

5.2. Query Parsing

In this section, we present the first phase of query processing corresponding to the XQuery Parser. We first describe XQGM and then present a general approach for converting XQuery queries into XQGM.

5.2.1. XML Query Graph Model (XQGM)

An intermediate query representation for processing queries over XML views of relational data needs to (a) be powerful enough to capture the full generality of a sophisticated query language such as XQuery, and (b) be amenable to a translation to SQL. We have designed the XQGM query representation with these in mind. XQGM is a natural extension of a SQL internal query representation called Query Graph Model (QGM) [35], which is used in a commercial relational database system. By building upon QGM in this manner, XQGM allows for a natural translation of XQuery queries to SQL. It also enables us to take the vast body of knowledge on relational query optimization and apply it to the XML query problem. In designing XQGM, we have also borrowed from the work on XML query algebras [22][75].

XQGM consists of a set of operators and functions that are designed to capture the semantics of an XML query. Table 5 shows the operators used in XQGM. As can be seen, the operators are a super-set of traditional relational operators. The select, project, join, group by, order by and union

OPERATOR	DESCRIPTION
Table	Represents a table in a relational database
Project	Computes results based on its input
Select	Restricts its input
Join	Joins two or more inputs
Groupby	Applies aggregate functions and grouping
Orderby	Sorts input based on column values
Union	Perfoms the union of two or more inputs
Unnest	Applies super-scalar functions to input
View	Represents a view
Function	Represents an XQuery function

Table 5: XQGM Operators

operators have the same semantics as their relational counterparts. The project operator is used to invoke functions (described later) in addition to projecting relational results. The table and view operators are used to refer to relational tables and XML view definitions respectively. The unnest operator is used to unnest XML lists. The function operator is used to invoke XQuery valued functions represented in XQGM.

The creation and manipulation of XML objects is done using XML functions. Table 6 presents a list of XML functions and also indicates the operators in which these functions can appear. We

now show how XQuery queries can be captured using the XQGM representation. We first provide illustrative examples, and then outline the general principles involved in the translation.

The XQGM graph for the view query given in Figure 67 is shown in Figure 70. Recall that the query produces a list of order XML elements, each of which has items and payments nested under it. We first explain the graph at a high level and then provide additional details.

	XML FUNCTION	DESCRIPTION	OPERATORS
1	cr8Elem(Tag, Atts, Clist)	Creates an element with tag name Tag, attribute list Atts, and contents Clist	Project
2	cr8AttList(A_1, …, A_n)	Creates a list of attributes from the attributes passed as parameters	Project
3	cr8Att(Name, Val)	Creates an attribute with name Name and value Val	Project
4	cr8XMLFragList(C_1, …, C_n)	Creates an XML fragment list from the content (element/text) parameters	Project
5	aggXMLFrags(C)	Aggregate function that creates an XML fragment list from content inputs	Groupby
6	getTagName(Elem)	Returns the element name of Elem	Project, Select
7	getAttributes(Elem)	Returns the list of attributes of Elem	Project, Select
8	getContents(Elem)	Returns the XML fragment list of contents (elements/text) of Elem	Project, Select
9	getAttName(Att)	Returns the name of attribute Att	Project, Select
10	getAttValue(Att)	Returns the value of attribute Att	Project, Select
11	isElement(E)	Returns true if E is an element, returns false otherwise	Select
12	isText(T)	Returns true if T is text, returns false otherwise	Select
13	unnest(List)	Superscalar function that unnests a list	Unnest

Table 6: XML Functions and the Operators in which they can appear

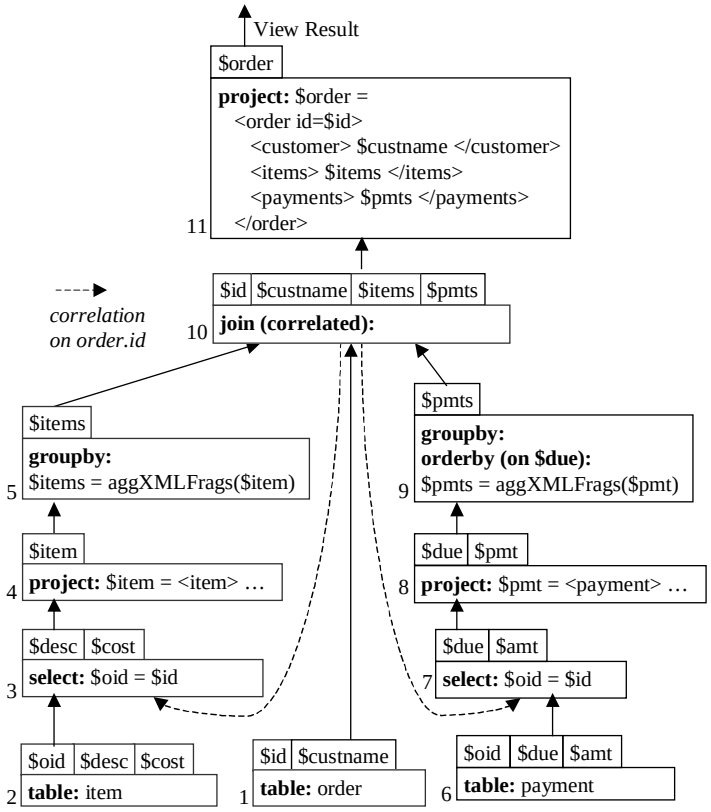

Figure 70: XQGM for the Order XML View

```
cr8Elem(order,
        cr8AttList(cr8Att(id, $id)),
        cr8XMLFragList(cr8Elem(customer,
                               cr8AttList(),
                               cr8XMLFragList($custname)),
                       cr8Elem(items,
                               cr8AttList(),
                               cr8XMLFragList($items)),
                       cr8Elem(payments,
                               cr8AttList(),
                               cr8XMLFragList($pmts))
                      )
         )
```

Figure 71: Expansion of Box 11 in Figure 70

Box 1 represents the *order* table with the two columns that are referenced in the query. Box 10 uses the notion of *correlated joins* (a join operator whose inputs are correlated sub-queries) to compute the list of items and payments associated with each order. Box 11 produces the order XML elements by tagging its inputs. While this tagging is shown as a template in box 11 for illustrative purposes, it is actually implemented as a call to the *cr8Elem* function as shown in Figure 71.

As shown in Figure 71, an *order* element has one attribute, *id*, created using the *cr8Att* function. In addition, an *order* element has three sub-elements, which are *customer*, *items* and *payments*. The *customer* element has no attributes but has a data value fragment represented by the input variable *$custname*. The *items* element again has no attributes but has a list of *item* sub-elements. This list of *item* sub-elements is represented by the input variable *$items*. The *payments* element is created similarly.

Returning to Figure 70, the correlated sub-query represented by boxes 6 through 9 computes the list of payments associated with each order. The payment rows from the *payment* table (box 6) that belong to an order are selected using a correlated predicate on the order id (box 7). Box 8 creates payment elements using the *cr8Elem* function and box 9 aggregates all the payment elements associated with an order into a list. For such grouping operations, the operator can be adorned

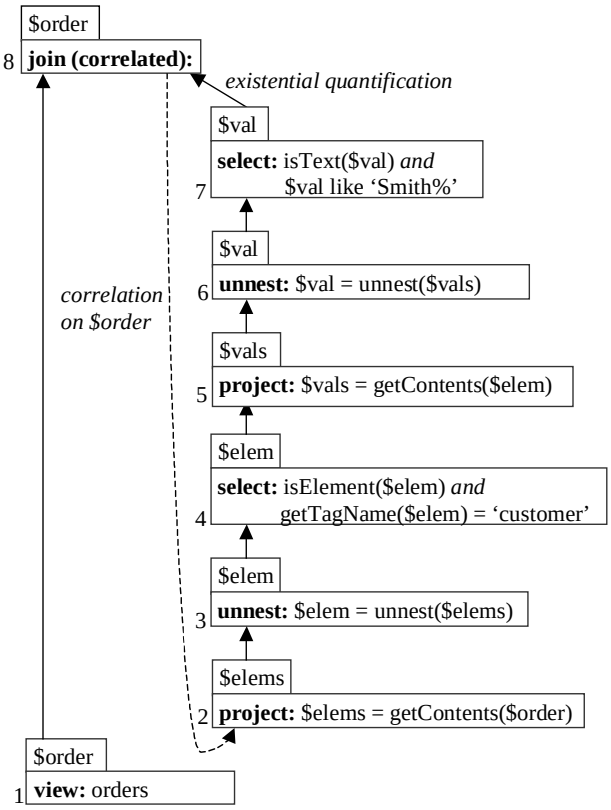

Figure 72: XQGM for Query over Order XML View

with an ordering condition that specifies the ordering of elements within the list. In our example, payments of an order are sorted by their due date. The computation of the list of items associated with each order (boxes 2-5) is similar to the computation of payments described above.

As another example of XQGM translation, Figure 72 shows the XQGM representation of the query shown in Figure 68. Box 1 captures the *orders* view referenced in the query. The where clause of the FLWR expression is computed as a correlated sub-query (boxes 2-7). This is done for the following reasons. Firstly, the where clause in XQuery can in general be an arbitrarily complex expression. Therefore, representing the where clause as a correlated query helps set up a separate context for its computation without inadvertently changing the cardinality or other properties of

the main query. Secondly, the where clause in XQuery has implicit existential semantics. In our example, the predicate ($order/customer/text() like 'Smith%') returns true if *any* customer element under order satisfies the predicate (if there exists more than one customer element). By representing the predicate as a correlated sub-query, the results can be existentially quantified (above box 7).

The XQGM representation of the predicate is straightforward. Box 2 gets the contents of a correlated order and box 3 unnests these contents. Box 4 selects only those contents that are sub-elements with tag name 'customer'. Box 5 gets the contents of these elements, box 6 unnests these contents, and box 7 selects the text content whose value satisfies the desired predicate.

XQuery also supports other complex expressions besides FLWR and sortby expressions. These include "let-eval", "if-then-else"" and "quantified" expressions. These expressions have clauses which themselves can be complex expressions. This generality leads to a powerful querying capability but increases the complexity of generating a semantically correct internal query representation. In our system, complex XQuery expressions are represented as correlated sub-queries with their separate context, much like how the where clause of the FLWR expression is represented in Figure 72. Sections 5.3 and 5.4 describe how this representation can be simplified using decorrelation and other query rewrite transformations.

5.3. View Composition

XML views with nested sub-elements are computed from flat relational tables. Navigational operations expressed as path expressions in XQuery queries traverse these nested structures to extract sub-elements and their attributes. Therefore, the query operators that traverse nested

structures effectively invert the query operators that create them in a view. Navigational operations can thus be eliminated by undoing the construction of the corresponding elements. Our view composition module performs this query simplification.

Removing all XML navigation operations offers several performance benefits. The obvious benefit is that the construction of intermediate XML fragments – those that do not appear in the final query result – is undone. Thus, only the desired XML fragments are materialized. As we shall shortly see, the other benefit of removing XML navigation functions is that it enables predicates and joins to be pushed down to the relational engine. As a result, there is no need for a full-fledged XML query-processor in the middleware layer. Only a space-efficient tagger, which will be described in more detail in Section 5.4, is required.

We have developed a complete set of composition rules involving XQGM functions that can be used to remove all XML navigation operations. We first present these rules and then discuss other query rewrite transformations that complement view composition.

5.3.1. Composition Rules

Functions 6 through 13 in Table 6 represent the functions that capture all the navigation operations in an XQuery query. Table 7 defines twelve composition rules that can be used to eliminate all occurrences of these navigational functions. These rules are complete in the sense that they specify how *all* occurrences of navigational functions can be eliminated. This is done by specifying a composition rule *for every possible input* to a navigational function, which specifies how the navigational function is to be removed for the given input. We now describe the composition rules in detail.

	FUNCTION	COMPOSES WITH	REDUCTION
1	getTagName	cr8Elem(Tag, Atts, Clist)	Tag
2	getAttributes	cr8Elem(Tag, Atts, Clist)	Atts
3	getContents	cr8Elem(Tag, Atts, Clist)	Clist
4	getAttName	cr8Att(Name, Val)	Name
5	getAttValue	cr8Att(Name, Val)	Val
6	isElement	cr8Elem(Tag, Atts, Clist)	True
7	isElement	Other than cr8Elem	False
8	isText	PCDATA	True
9	isText	Other than PCDATA	False
10	unnest	aggXMLFrags(C)	C
11	unnest	cr8XMLFragList(C_1, …, C_n)	$C_1 \cup \ldots \cup C_n$
12	unnest	cr8AttList(A_1, …, A_n)	$A_1 \cup \ldots \cup A_n$

Table 7: Composition Rules

Rule 1 in Table 7 says that the *getTagName* function when applied to the *cr8Elem* can be reduced to the first argument of the *cr8Elem* function (which is the tag name of the created element). Rules 2-5 are defined in a similar manner. Rules 6 and 7 replace the *isElement* function by *true* or *false*, depending on whether the input is an element or not. Rules 8 and 9 are defined similarly. Rule 10 composes the *unnest* function with the *aggXMLFrags* function by simply returning the input to *aggXMLFrags* without performing any aggregation. Rule 11 composes the *unnest* function with the *cr8XMLFragList* function by reducing the *unnest* function to a union of all the arguments of the *cr8XMLFragList* function. Rule 12 is defined similarly.

5.3.2. Applying Composition Rules

We eliminate all navigational operations by repeated application of the composition rules shown in Table 7. This step is complemented by a number of other query rewrite transformations that push down predicates and remove unreferenced columns and operators. We illustrate this step in view composition using an example.

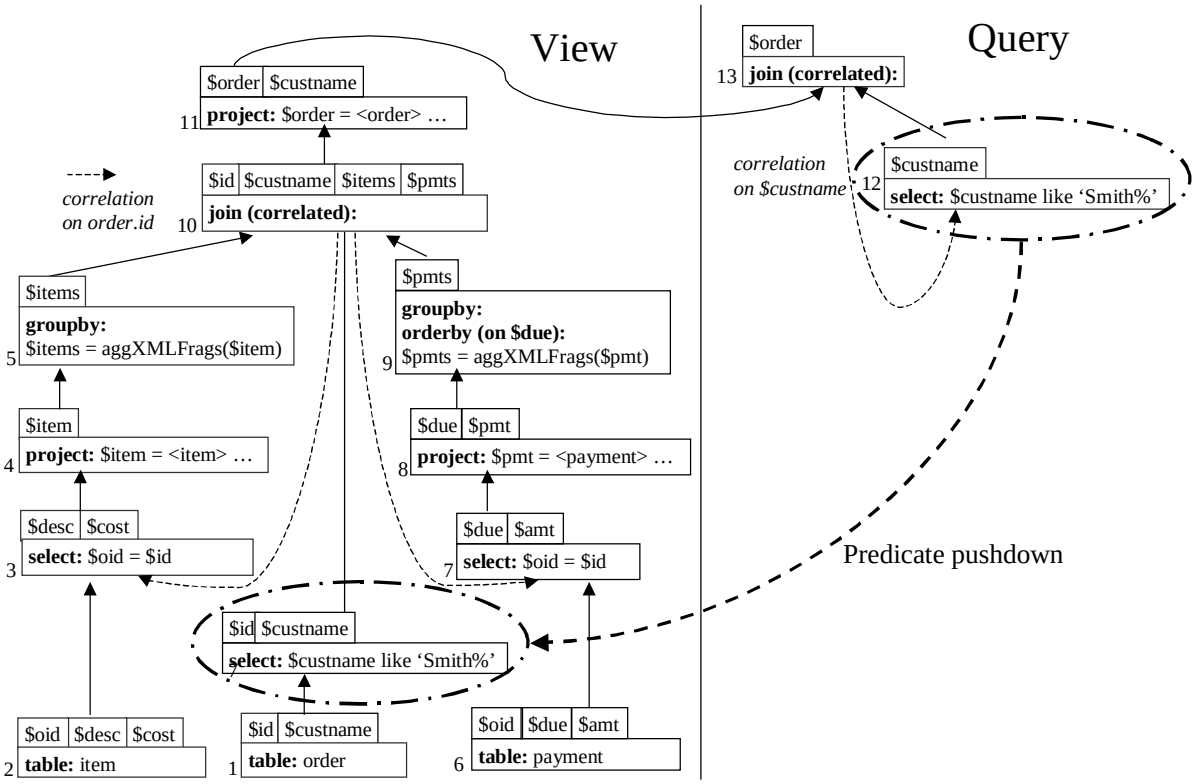

Figure 73: XQGM after View Composition

The result of composing the query graph in Figure 72 with the view graph in Figure 70 is shown in Figure 73. As can be seen, all XML navigation functions in Figure 72 have been composed with their counterparts in box 11 of Figure 70 (box 11 is also expanded in Figure 71). As a result, the selection predicate is specified directly over *$custname* (box 12 in Figure 73). Once navigational functions have been removed, standard query rewrite transformations such as predicate pushdown can be applied. In our example, the predicate has been pushed down to a select box immediately above the *order* table.

Although not illustrated by the above example, all intermediate XML fragments are also removed at this stage of query processing. For example, if the query in Figure 68 had just selected the items in the desired orders, only the XML construction functions that produce items would be present in the output. Since the order elements as a whole would no longer be referenced, they would have been removed from the query graph and hence, would not have been materialized.

5.4. Computation Pushdown

The goal in this phase of query processing is to push all data and memory intensive operations down to the relational engine as an efficient SQL query. We describe two query processing techniques that make this possible.

5.4.1. Query Decorrelation

In Section 5.2.1, we showed how complex expressions in XQuery are represented using correlations. However, as shown in the previous chapter, executing XML queries as correlated queries over a relational database leads to poor performance. We thus present query decorrelation [64] as a necessary step for efficient XML query execution.

The result of decorrelating the XQGM of Figure 73 is shown in Figure 74. As shown, the construction of the list of items (boxes 2-5) associated with orders has been decorrelated. This is done by directly joining the selected orders (box 10) with the item table, instead of performing a correlated selection condition. The items are then tagged (box 4) and grouped on the id of the order (box 5) to create a list of items for each selected order. The list of payments associated with each purchase order is computed similarly (boxes 6-9).

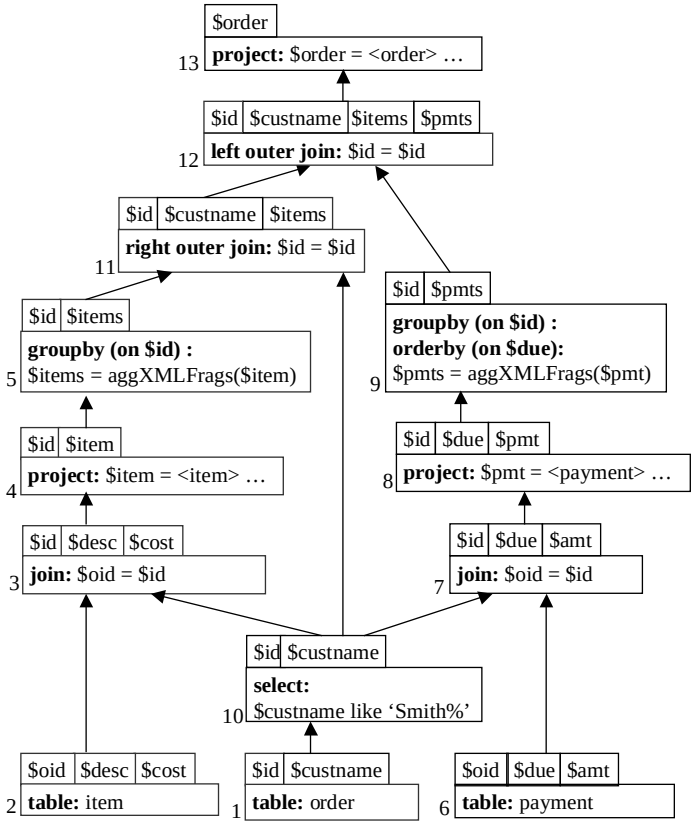

Figure 74: XQGM after Decorrelation

Once the list of items associated with orders is computed, these are outer joined with the selected orders (box 11). An outer join is necessary to preserve all selected orders because the join in box 3 would have eliminated orders without any items. This outer joined result is similarly outer joined with payments (box 12).

5.4.2. Tagger Pullup

After query decorrelation, the next step is to generate a SQL query that can efficiently produce the relational content for constructing the result XML document. As shown in Chapter 4, the "sorted outer union" SQL query is one of the most efficient and stable techniques for this purpose. However, the generation of the sorted outer union SQL query from the XQGM graph is

OPERATOR	USAGE	FUNCTIONS
Merge	Merges ordered streams	cr8Elem, cr8Att, cr8XMLFragList, cr8AttList
Aggregate	Computes aggregates	AggXMLFrags
Union	Unions ordered streams	
Input	Manages relational rows	

Table 8: Summary of Tagger Operators

complicated by the fact that the "tagger operations", which construct XML fragments, can be mixed with the "SQL operations", which join or otherwise manipulate relational content (see Figure 74). As a result, the tagger and SQL operations need to be separated before the sorted outer union SQL query can be generated. This process of separation is what we call "tagger pull-up".

During tagger pull-up, relational operations are pushed to the bottom of the graph, and XML construction functions are pulled up to the top of the query graph. The bottom portion of the query graph is then converted to a sorted outer union SQL statement and sent to the relational engine for execution. The top portion is transformed into a "tagger run-time" graph, which produces the result XML documents.

The tagger run-time graph consists of a set of tagger operators. In contrast to the relational operators, which are designed for execution by a sophisticated relational engine, the tagger operators are designed for efficient in-memory processing in the middleware. The tagger operators process ordered streams of rows and produce the XML result in constant space, and in a single pass over the SQL query results. The list of tagger operators, along with the functionality of each operator, is shown in Table 8. The XML construction functions that can be implemented in each tagger operator are also shown.

Our system implements a general tagger pull-up algorithm that works for complex query graphs. There are, however, some queries for which all data and memory intensive computation cannot be pushed down to the relational engine. More details regarding this are presented in Section 5.6, along with a discussion of how this limitation could be removed.

5.4.2.1. An Illustrative Example

An example of tagger pull-up is shown in Figure 75. This graph is the result of applying tagger pull-up transformations to the query graph in Figure 74. As can be seen in Figure 75, the bottom part consists of SQL statements, while the top part consists of tagger operators. For the rest of this section, we describe how the tagger operators produce the XML result in a single pass over the SQL results. Details on separating the SQL part from the tagger part are deferred until the next section.

The bottom part of Figure 75 produces three SQL streams. The middle stream produces the selected orders, ordered by their id. The first and third streams produce the desired items and payments, respectively, ordered by the id of the order they are associated with. The third stream is also ordered by the due date to capture the ordering of payments. Since all the results are ordered using the order id, the result XML elements can be constructed in a single pass over the SQL results. This is done by the tagger operators, which work as follows.

When an order flows up through the tagger input operator (box 1), only the items and payments corresponding to that order are let through by their corresponding tagger input operators (boxes 2 and 5). These items and payments are then tagged (boxes 3 and 6) and grouped (boxes 4 and 7).

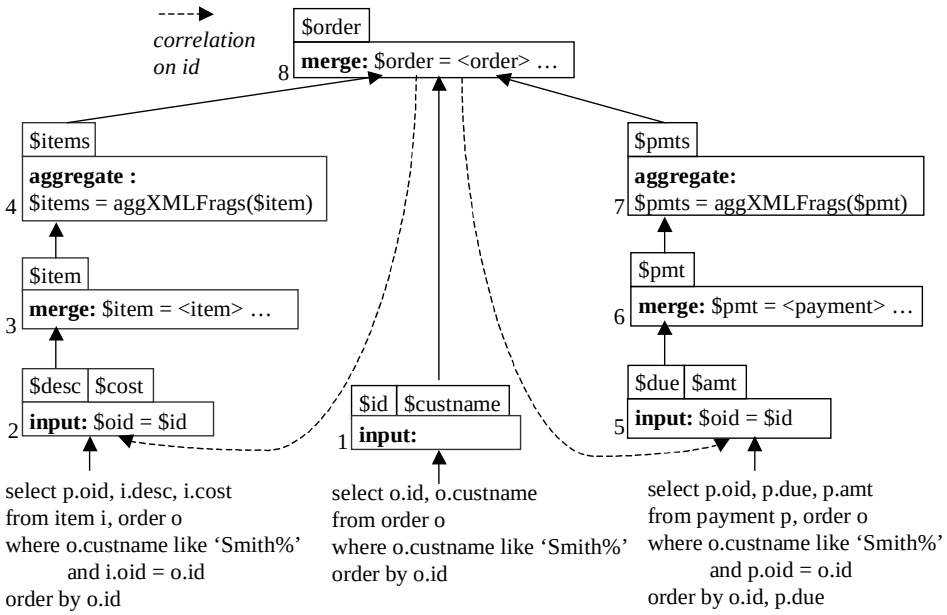

Figure 75: XQGM after Tagger Pull-up

The top tagger operation (box 8) then merges these together to produce the result. This process is repeated for the next order.

Although Figure 75 shows three SQL statements, these are actually executed as a single SQL statement by performing an outer union. This is done using the sorted outer union technique proposed and evaluated in Chapter 4.

5.4.2.2. Tagger Pull-up Transformations

We now describe the XQGM transformations used during tagger pull-up. These transformations work in a bottom-up fashion on the initial query graph (such as Figure 74). Each transformation matches a fragment of the query graph and converts it to a semantically equivalent fragment in which the SQL part is separated from the tagger part. Repeated applications of these transformations produce the final separated query graph (such as Figure 75).

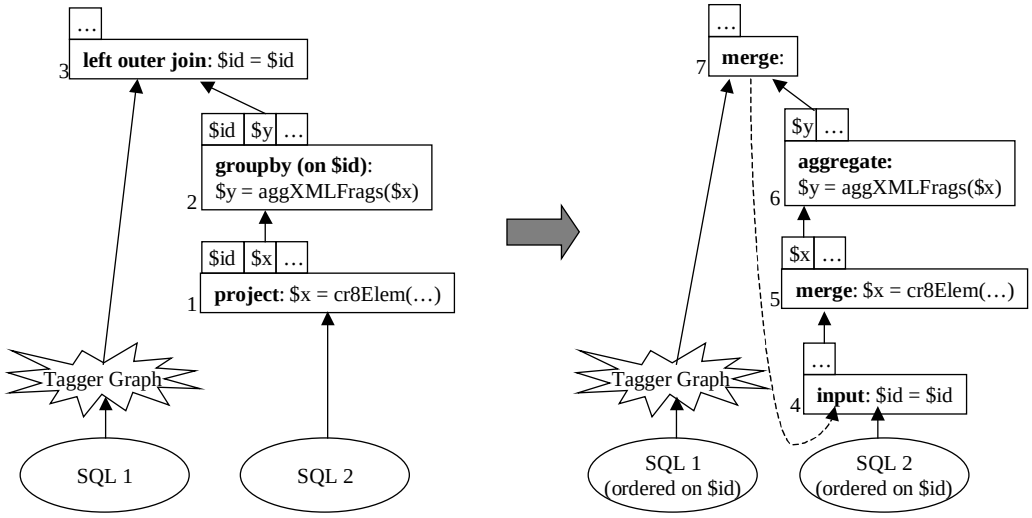

Figure 76: Tagger Pull-up Transformation

Figure 76 shows the XQGM transformation that implements tagger pull-up for nested XML structures. The left side of the figure shows the XQGM fragment before the transformation. As can be seen, tagged elements are created (box 1), grouped into a list (box 2), and joined with their parent (box 3). This pattern would match boxes 4, 5 and 11 in Figure 10 (note that a left outer join is the same as a right outer join with the inputs flipped).

The right side of the transformation in Figure 76 shows the result of tagger pull-up. The merge (box 5) and aggregate (box 6) operators are used to create and group the nested elements, respectively. A merge operator (box 7) is used to relate the list of elements to their parent. A merge is sufficient for this purpose (instead of the join used earlier) because an ordering condition on parent id is pushed down to the SQL queries. A tagger input operator (box 4) is used to gate the children that belong to a certain parent, as described earlier.

The result of applying this transformation to boxes 4, 5 and 11 in Figure 74 produces boxes 3, 4 and 8 in Figure 75. A repeated application of this transformation would transform boxes 8, 9 and 12 in Figure 74 to boxes 6, 7 and 8 in Figure 75 (the duplicate box 8's created are merged using other query graph simplification rules).

5.5. Implementation and Performance

We have implemented all of the techniques described above as part of the XPERANTO middleware system [16]. Our implementation is in Java and uses JDK 1.2. We use JDBC as the API to connect to a relational database system. As a result, our implementation works on top of most commercial database systems including DB2, Oracle and Microsoft SQL Server.

Based on our prototype implementation, we have evaluated the performance of our proposed techniques. There are two factors that characterize the performance of queries in our context. The first is query compilation time, which consists of the time spent parsing the query, performing view composition, generating the SQL query and setting up the tagger run-time graph. The second is query execution time, which consists of the time spent executing the SQL query and tagging the SQL results to produce the output XML document. Chapter 4 has evaluated query execution performance and has shown the superiority of the sorted outer union SQL plans that we generate. Hence, we only focus on query compilation time here.

Our experiments were performed using a 600 MHz Pentium III processor with 256MB of main memory running Windows NT 4.0. We used DB2 version 7.2 as our database system. We ran XPERANTO and the database system on the same machine to avoid unpredictable network

delays. We considered queries that accessed up to 5 XML views. Each XML view nests 3 relational tables in a manner similar to Figure 67.

The compilation time for our experimental queries was always less than half a second. For instance, the compilation time for a query that accessed 15 relational tables through 5 XML views was about 400 milliseconds. It takes even less time to compile queries that access fewer views. As a result, there is a very small compile-time overhead associated with performing the optimizations proposed in this chapter.

It is important to note that this small compile-time overhead is more than offset by the associated performance gains. This is due to two reasons. The first reason is that the view composition module eliminates the need to materialize intermediate XML fragments that do not appear in the final query result. As a result, only the relevant data is fetched from the relational engine. Thus, for typical queries that select only a small subset of data in an XML view, this results in many orders of magnitude improvement in performance. As a simple example, consider an XML view that publishes one million available items. If the user want details on only one of these items, it is clear that retrieving only the desired item will be orders of magnitude better than materializing all one million items and then selecting the desired one. This advantage is especially relevant when the underlying relational data changes often and cannot be easily cached in the middleware layer.

The other significant performance benefit is due to the computation pushdown module. By effectively harnessing the relational engine to process large parts of XML queries, it eliminates the need for a full-fledged XML processor in the middleware layer – only a small, space-efficient tagger run-time mechanism is required. This is important because no existing native XML query

processor has performance characteristics that are comparable to that of a parallel, scalable relational engine.

5.6. Limitations of the Approach and Possible Solutions

In this chapter, we have focused on the problem of efficiently evaluating XML queries over XML views of relational data. In this context, we have described a computation pushdown mechanism that allows all data and memory intensive computation to be pushed down to the underlying relational engine as a SQL query. However, as alluded to earlier in the chapter, there are certain XML queries that cannot be directly pushed down to the relational engine. The first class of such queries are meta-data queries. These queries span relational meta-data (column and table names) and data (column values). While XQuery can naturally express such queries, SQL cannot. This is because SQL does not have certain higher-order operators [45]. Fortunately, it turns out that the desired higher-order operators can be provided in the middleware while still pushing most computation down to the relational engine (see [67] for more details).

The second class of queries that cannot be directly pushed down as SQL are those that perform user-defined operations on intermediate XML fragments. For example, consider a query that joins department and employee XML fragments using a user-defined XML predicate such as deptcontains(deptFrag, empFrag). It is important to note that the join predicate here involves XML fragments, and is not a predicate on basic data types such as integers (joins on basic data types can be handled using our computation push down mechanism). The reason that the join on XML fragments cannot be pushed down is because the relational engine does not know about

XML fragment construction. A similar problem occurs when trying to order or group on XML fragments.

One solution is to perform these operations outside the relational engine, but this requires the duplication of sophisticated relational functionality, such as joins and sorts. Another solution, and the one we advocate, is to add primitives to construct XML document fragments inside the relational engine. In this way, all data and memory intensive processing can be done inside the relational engine. As shown in Chapter 4, the most efficient way to construct XML fragments inside the engine is to use the sorted outer union query plan. Integrating the computation pushdown technique with the relational engine so that these plans can be automatically generated is an area for future investigation.

CHAPTER 6: RELATED WORK

In this chapter, we describe work that is related to the content of this dissertation.

6.1. Storage and Querying of XML Documents

There has been considerable interest in developing special-purpose database systems for storing and querying XML documents [49][52]. Many of the abstracts submitted to the XML query languages workshop use this approach [58]. Our goal in this dissertation, however, has been to investigate the use of relational database systems to process queries on XML documents.

There have been other proposals for storing and querying XML documents using a relational database system. One such approach is STORED [28], which uses a combination of relational and semi-structured techniques to process arbitrary XML documents. In contrast, we begin with the assumption that the document conforms to a schema and store the document entirely within the relational system. Further, unlike STORED, we handle recursive queries and address the issue of constructing the result in XML.

Another approach to storing and querying XML documents using a relational database system has been proposed in [33]. In this approach, an XML document is viewed as a graph, with the nodes of the graph representing XML elements and attributes, and the edges of the graph representing parent-child relationships. Edges of this graph are stored as rows in a table. No schema information of XML documents is used in the creation of edge table. Queries over the XML document are converted to SQL queries over the edge table. Since each parent-child relationship is

stored as a separate row in the table, a join operation is required for every step of a path expression in an XML query. The main difference between this approach and our approach is that we aggressively exploit schema information of XML documents in creating relational tables. By doing so, we use techniques such as inlining in order to reduce the number of expensive join operations required to answer XML path expression queries.

Oracle's relational database system provides some basic support for querying XML documents [53]. However, the translation from XML document schemas to relational schemas is manual and not automatic as in our approach. In addition, Oracle does not support XML queries over XML documents (only SQL queries over the tables are supported).

IBM's DB2 relational database system can also store and query XML documents [19]. However, unlike our approach, DB2 does not shred XML documents into tuples in tables. As a result, the DB2 relational engine cannot be harnessed for processing complex queries over XML documents because the structure of the XML document CLOB is opaque to the relational query processor. Consequently, DB2 provides only limited query support for XML documents (the query language supported in XPath, which cannot specify joins, for instance).

Microsoft's SQL Server relational database system can store XML documents as CLOBs. However, unlike our approach, there is no facility to query these XML documents using an XML query language. Only SQL extensions, which do not have the full flexibility of an XML query language, can be used to query XML documents.

There has been some work done on using object-oriented database systems for storing and querying SGML documents [20]. The conclusion was that this is feasible with some extensions to object-oriented query languages. Our work considers a more restricted set of documents (XML, rather than SGML) and considers mapping to the relational model, rather than a general object-oriented model.

6.2. Materializing Relational Data as XML Documents

In our work on materializing relational data as XML documents we (a) proposed a SQL-based query language for specifying the conversion from relations to XML, and (b) investigated various execution strategies for efficiently materializing relational data as XML documents. We now describe work related to both these contributions.

6.2.1. Query Languages for Publishing Relational Data as XML

In addition to the SQL extensions proposed in this dissertation, there are other language proposals for specifying the conversion from relational data to XML documents [8][31][51]. SilkRoute [31] uses a combination of SQL and XML-QL, to specify the construction of XML documents. Microsoft Corporation uses XDR Schemas [51], which are annotated XML Schemas [77] for specifying the mapping from relational data to the desired XML document structure. Oracle Corporation [8] 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.

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 [72] types in order to publish relational data as XML documents.

Query languages for Non First Normal Form (NF^2) database systems also 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. 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) [26][57][59]. Further, since NF^2 database systems do not deal with tags, their query languages [44][62] cannot specify user-defined tagging of XML documents.

6.2.2. Efficiently Materializing Relational Data as XML

Fernandez et al. [32] 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 [32] 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 [32]) 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 [32] 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 [32] 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.).

The work on NF^2 database systems, dealing with the storage and querying of nested objects, is related to our work on materializing nested XML documents. There are, however, some key differences. Firstly, 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 [26][62], 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.

The work on assembling complex objects in object-oriented database systems [41][71] is also related to our work on materializing relational data as complex XML documents. 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 XML elements from relational data.

6.3. Querying XML Views of Relational Data

Most major commercial database systems provide a way to create XML views of relational data [8][19][51]. However, in contrast to the approach presented in this dissertation, most of these systems do not support queries over XML views [8][19]. Microsoft's SQL Server is the only one that supports queries over XML views, but this query support is very limited. This is because queries are specified using XPath [76], which is a subset of the XQuery query language we consider. For instance, unlike XQuery, XPath cannot specify joins.

XML-based data integration systems [10][11][22][54] also wrap relational data sources as XML views. In this context, there has been work done on pushing part of XML query execution down to the relational sources, with the remainder of the query being executed by an XML run-time engine in the integration layer [22]. In contrast to this approach, we support a general XML query capability over relational databases without the need for a full-blown XML run-time engine. Our approach also differs in that we generate query plans that are optimized for relational databases. This is important because, as shown in Chapter 4, the choice of query plans can make a significant difference in performance, especially when constructing complex XML results. These differences apart, our system can indeed be used as an efficient XML wrapper for relational sources.

SilkRoute [31][32] is a related system that supports queries over XML views of relational data. Our work differs from SilkRoute in the following respects. Firstly, we support a more powerful query facility based on XQuery. XQuery has features like arbitrarily nested expressions and nested order, which are not supported in SilkRoute. Secondly, we use a view composition technique that is complete and yet produces minimal SQL queries. This is in contrast to SilkRoute's view

composition mechanism, which produces SQL queries with redundant predicates and joins (these queries can be minimized, but this problem is NP-complete [32]). Thirdly, our computation pushdown mechanism is more general because it takes into account the greater flexibility of XQuery. Finally, unlike SilkRoute, our internal XML query model naturally extends the relational model. Our work can thus serve as the basis for extending relational databases with XML features.

There has been some work on providing XML views over native XML repositories, notably Lore [49]. Most of the work in this category, however, has not been on efficiently answering queries over views but on incremental view maintenance [2][6] and on rewriting queries using materialized views [55]. The latter stems from the fact that they take a fundamentally different approach from the one we are taking; they assume the existence of an XML query processor that can make use of materialized XML views, while we push query processing down to the relational engine by view rewriting.

The large body of work on object views [1][15][36][42][43][61] is also related to our work on XML views because both essentially deal with complex graph structures. On the specific subject of query rewriting, however, there is less literature available [23]. The one most relevant to XML view composition by query rewriting is the work in [14], where unnecessary nesting operations are removed during view composition. However, their algorithm proposed for object views does not deal with many XML specific features that we address, such as general path expressions, tag variables, and flexible typing.

There has been some published work on providing object views of relational sources [9][40][60]. Unlike our work on XML view composition, however, none of this work has considered the problem of composing a hierarchy of object views by query rewriting.

CHAPTER 7: CONCLUSION

XML is rapidly emerging as the standard format for data representation and exchange over the Internet. This has created a new set of data management requirements involving XML. However, for the foreseeable future, relational database systems will continue to be used for most data management tasks because of their reliability, scalability, performance and tools. Therefore, a bridge is needed to satisfy the requirements of Internet-based XML applications while still leveraging relational database technology. In this dissertation, we have presented various techniques for bridging relational technology and XML.

7.1. Contributions

The first contribution of this dissertation is a technique for storing and querying XML documents using a relational database system. This technique establishes that it is indeed possible to use relational database systems for querying XML documents. This technique also shows how XML schema information can be exploited to speed up queries over XML documents stored in a relational database system. The key to this is what we call *inlining*, which minimizes the number of relational join operations required to process XML queries. Our evaluation using 37 real DTDs shows that the inlining technique is effective in many cases. There are some cases, however, where relational database systems are not likely to perform well. We have identified the causes for these limitations and have proposed extensions to the relational model that can help address this problem.

The second contribution of this dissertation is a way to materialize XML documents from data stored in relational database systems. In this context, we have proposed an extension to the SQL query language for specifying the conversion from relations to XML. We have also proposed, categorized and evaluated various techniques for efficiently materializing relational data as XML documents. In this context, we have shown that the *outer union plans* offer up to an order of magnitude improvement in performance over the naïve, more commonly used approaches.

The final contribution of this dissertation is a technique for querying XML views of relational data. In this context, we have developed a general framework for processing arbitrarily complex queries over XML views of relational data. We have also developed novel *view composition* and *computation pushdown* algorithms for efficiently evaluating XML queries over XML views of relational data. These algorithms push all data and memory intensive computation in an XML query down to the relational engine for efficient processing.

7.2. Possibilities for Future Work

There are many interesting avenues for future work. Regarding storing and querying XML documents, our experience suggests that extending relational database systems for this purpose can lead to significant benefits. We have identified a list of possible extensions to relational systems in Section 3.4. Implementing and evaluating these extensions will increase the effectiveness of relational systems for processing XML queries, both in terms of functionality and in terms of performance.

In the area of efficiently materializing relational data as XML documents, the issue of constructing recursively structured XML documents is an open research problem. Other open issues include

studying the impact of parallelism when constructing large XML documents, and the design and analysis of efficient memory-management techniques to extend the useful range of the unsorted outer union plans.

Regarding querying XML views of relational data, various optimization strategies can be applied in addition to those proposed in this dissertation. For example, rather than re-constructing the result XML documents from the base relations every time, relevant fragments of the XML documents can be cached. Another possibility for optimization is to push XML document construction down to "XML aware" relational database systems. In addition to optimization opportunities, there is also scope to provide important new functionality. One such issue is updating XML views of relational data using an XML query language, and having these updates trickle down to the relational tables. Another issue is efficiently evaluating XML queries with order using a relational database system.

By providing a way to store and query XML documents using a relational database system, this dissertation has also set the stage for a unified data management system. If both XML documents and existing relational data can be stored and queried using the same query-processing framework (i.e., a relational database system), then it should be able to query across the two. Developing a system architecture that can such support seamless querying over both relational data and XML documents is an open research problem.

BIBLIOGRAPHY

[1] S. Abiteboul, A. Bonner, "Objects and Views," Proceedings of the ACM SIGMOD Conference on Management of Data, Colorado, 1991.

[2] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, Y. Zhuge, "Views for Semistructured Data," Proceedings of the Workshop on Management of Semistructured Data, Tucson, Arizona, 1997.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, "The Lorel Query Language for Semistructured Data", International Journal on Digital Libraries, 1(1), April 1997, pp. 68-88.

[4] S. Abiteboul, "On Views and XML," Proceedings of the ACM Symposium on Principles of Database Systems (PODS), Pittsburgh, 1999.

[5] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, "Active Views for Electronic Commerce," Proceedings of the Conference on Very Large Data Bases (VLDB), Edinburgh, Scotland, 1999.

[6] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, J. L. Wiener, "Incremental Maintenance for Materialized Views over Semistructured Data," Proceedings of the Conference on Very Large Data Bases (VLDB), New York, USA, 1999.

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

[8] 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.

[9] T. Barsalou, A. M. Keller, N. Siambela, G. Wiederhold, "Updating Relational Databases through Object-Based Views," Proceedings of the SIGMOD Conference, Colorado, 1991.

[10] C. Baru, "XViews: XML Views of Relational Schemas," International Workshop on Internet Data Management, Florence, Italy, 1999.

[11] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, V. Chu, "XML-based Information Mediation with MIX," Proceedings of the ACM SIGMOD Conference on Management of Data, Pittsburgh, 1999.

[12] C. Baru, B. Ludaescher, Y. Papakonstantinou, P. Velikhov, V. Vianu "Features and Requirements for an XML View Definition Language: Lessons from XML Information Mediation", Position paper in W3C's Query Language Workshop, 1998.

[13] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, "A Query Language and Optimization Techniques for Unstructured Data", Proceedings of the ACM SIGMOD Conference on Management of Data, Montreal, Canada, June 1996.

[14] M. Carey, J. Kiernan, "Query Rewrite Transformations for Nested Set Elimination in Views," patent pending.

[15] M. Carey, S. Rielau, B. Vance, "Object View Hierarchies in DB2 UDB," Proceedings of the International Conference on Extending Database Technology (EDBT), Konstanz, Germany, 2000.

[16] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, S. Subramanian, "XPERANTO: Publishing Object-Relational Data as XML", Workshop on Web and Databases (WebDB), Dallas, Texas, May 2000.

[17] M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, S. Subramanian, "XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents", Demonstration at the Conference on Very Large Data Bases (VLDB), Cairo, Egypt, September 2000.

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

[19] J. Cheng, J. Xu, "XML and DB2", Proceedings of the International Conference on Data Engineering (ICDE), San Diego, March 2000, pp. 569-573.

[20] V. Christophides, S. Abiteboul, S. Cluet, M. Scholl, "From Structured Documents to Novel Query Facilities", Proceedings of the ACM SIGMOD Conference on Management of Data, Minneapolis, Minnesota, May 1994.

[21] V. Christophides, S. Cluet, G. Moerkotte, "Evaluating Queries with Generalized Path Expressions," Proceedings of the ACM SIGMOD Conference on Management of Data, Montreal, Quebec, Canada, 1996.

[22] V. Christophides, S. Cluet, J. Simeon, "On Wrapping Query Languages and Efficient XML Integration", Proceedings of the ACM SIGMOD Conference on Management of Data, Dallas, May 2000, pp. 141-152.

[23] S. Cluet, C. Delobel, "A General Framework for the Optimization of Object-Oriented Queries," Proceedings of the ACM SIGMOD Conference on Management of Data, San Diego, 1992.

[24] G. Copeland, S. Khoshafian, "A Decomposition Storage Model", Proceedings of the ACM SIGMOD Conference on Management of Data, Austin, Texas, May 1985.

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

[26] 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.

[27] 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.

[28] A. Deutsch, M. Fernandez, D. Suciu, "Storing Semi-structured Data with STORED", Proceedings of the ACM SIGMOD Conference on Management of Data, Philadelphia, Pennslyvania, May 1999.

[29] R. Fagin, "Multi-valued Dependencies and a New Normal Form for Relational Databases", ACM Transactions on Database Systems, 2(3), pp. 262-278, 1977.

[30] M. Fernandez, D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas", Proceedings of the International Conference on Data Engineering (ICDE), Orlando, Florida, February 1998.

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

[32] 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, pp. 103-114.

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

[34] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, J. Widom, "The TSIMMIS Approach to Mediation: Data Models and Languages," Journal of Intelligent Information Systems, Vol. 8, No. 2, 1997.

[35] L. Haas, J. Freytag, G. Lohman, H. Pirahesh, "Extensible Query Processing in Starburst", Proceedings of the ACM SIGMOD Conference on Management of Data, Portland, May 1989, pp. 377-388.

[36] S. Heiler, S. Zdonik, "Object Views: Extending the Vision," Proceedings of the International Conference on Data Engineering (ICDE), Los Angeles, USA, 1990.

[37] IBM Corporation, XML Parser for Java, http://www.alphaworks.ibm.com/tech/xml4j.

[38] IBM Corporation, XML Lightweight Extractor, http://www.alphaworks.ibm.com/tech/xle.

[39] G. Jaeschke, H. J. Schek, "Remarks on the Algebra of Non First Normal Form Relations", Proceedings of the ACM Symposium on Principles of Database Systems (PODS), Los Angeles, California, March 1982.

[40] A. Kawaguchi, D. Lieuwen, I. Mumick, K. Ross, "Implementing Incremental View Maintenance in Nested Data Models," Workshop on Database Programming Languages, Colorado, USA, 1997.

[41] 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.

[42] M. Kifer, W. Kim, Y. Sagiv, "Querying Object-Oriented Databases," Proceedings of the ACM SIGMOD Conference on Management of Data, California, 1992.

[43] W. Kim, W. Kelley, "On View Support in Object-Oriented Database Systems," Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press and Addison-Wesley, 1995.

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

[45] L. Lakshmanan, F. Sadri, I. Subramanian, "SchemaSQL – A Language for Interoperatability in Relational Multi-Database Systems", Proceedings of the Very Large Data Bases Conference, Mumbai, India, Sep. 1996, pp. 239-250.

[46] B. Ludascher, Y. Papakonstantinou, P. Velikhov, V. Vianu, "View Definition and DTD Inference for XML," Post-ICDT Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, 1999.

[47] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, D. Olteanu, "XML and Relational: How to Live with Both", Demonstration at the Very Large Data Bases (VLDB) Conference, Cairo, Egypt, September 2000.

[48] I. Manolescu, D. Florescu, D. Kossmann, "Answering XML Queries over Heterogeneous Data Sources", Proceedings of the Conference on Very Large Data Bases (VLDB), Rome, Italy, September 2001 (to appear).

[49] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, "Lore: A Database Management System for Semistructured Data", SIGMOD Record, 26(3), pp. 54-66, September 1997.

[50] J. McHugh, J. Widom, "Compile-Time Path Expansion in Lore", Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Israel, January 1999.

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

[52] J. Naughton, D. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, E. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, R. Chen, "The Niagara Internet Query System", IEEE Data Engineering Bulletin, Vol. 24, No. 2, pp. 27-33, 2001.

[53] Oracle Corporation, "XML Support in Oracle 8 and beyond", Technical white paper, http://www.oracle.com/xml/documents.

[54] Y. Papakonstantinou, S. Abiteboul, H. Garcia-Molina, "Object Fusion in Mediator Systems," Proceedings of the Very Large Data Bases (VLDB) Conference, Mumbai, India, 1996.

[55] Y. Papakonstantinou, V. Vassalos, "Query Rewriting for Semistructured Data," Proceedings of the ACM SIGMOD Conference on Management of Data, Pittsburgh, 1999.

[56] 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.

[57] 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.

[58] The Query Languages Workshop (QL'98), http://www.w3.org/TandS/QL/QL98/, December 1998.

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

[60] M. Rys, M. C. Norrie, H. Schek, "Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System," Proceedings of the Very Large Data Bases (VLDB) Conference, Mumbai, India, 1996.

[61] C. Santos, S. Abiteboul, C. Delobel, "Virtual Schemas and Bases," Proceedings of the International Conference on Extending Database Technology (EDBT), Cambridge, United Kingdom, 1994.

[62] 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.

[63] T. Sellis, "Multiple-Query Optimization", ACM Transactions on Database Systems, 12(1), pp. 23-52, June 1990.

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

[65] 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, Edinburgh, Scotland, 1999.

[66] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, B. Reinwald, "Efficiently Publishing Relational Data as XML Documents", Proceedings of the Very Large Data Bases (VLDB) Conference, Cairo, Egypt, Sep. 2000, pp. 65-76.

[67] J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, J. Naughton, "XPERANTO: Bridging Relational Technology and XML", IBM Research Report, June 2001.

[68] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, B. Reinwald, "Efficiently Publishing Relational Data as XML Documents", VLDB Journal, 2001 (to appear).

[69] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, J. Funderburk, "Querying XML Views of Relational Data", Proceedings of the Conference on Very Large Data Bases (VLDB), Rome, Italy, September 2001 (to appear).

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

[71] 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.

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

[73] I. Tatarinov, Z. Ives, A. Halevy, D. Weld, "Updating XML", Proceedings of the ACM SIGMOD Conference on Management of Data, Santa Barbara, California, May 2001.

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

[75] World Wide Web Consortium, "The XML Query Algebra", W3C Working Draft, 2001.

[76] World Wide Web Consortium, "XML Path Language (XPath) Version 1.0", W3C Recommendation, November 1999. See http://www.w3c.org/TR/xpath.html.

[77] 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.

[78] World Wide Web Consortium, "XML Query Requirements", W3C Working Draft, August 2000. See http://www.w3c.org/TR/xmlquery-req.

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

[80] World Wide Web Consortium, "XSL Transformation (XSLT) Version 1.0", W3C Recommendation, November 1999. See http://www.w3c.org/TR/xslt.html.

[81] C. Zaniolo, "The Database Language GEM", Proceedings of the ACM SIGMOD Conference on Management of Data, San Jose, California, May 1983.