Storing and Querying Ordered XML Using a Relational Database System

Igor Tatarinov*
University of Washington

Stratis D. Viglas*
University of Wisconsin

Kevin Beyer
IBM Almaden Research Center

Jayavel Shanmugasundaram*
Cornell University

Eugene Shekita
IBM Almaden Research Center

Chun Zhang*
University of Wisconsin

ABSTRACT

XML is quickly becoming the *de facto* standard for data exchange over the Internet. This is creating a new set of data management requirements involving XML, such as the need to store and query XML documents. Researchers have proposed using relational database systems to satisfy these requirements by devising ways to "shred" XML documents into relations, and translate XML queries into SQL queries over these relations. However, a key issue with such an approach, which has largely been ignored in the research literature, is how (and whether) the *ordered* XML data model can be efficiently supported by the unordered relational data model. This paper shows that XML's ordered data model can indeed be efficiently supported by a relational database system. This is accomplished by encoding order as a data value. We propose three *order encoding methods* that can be used to represent XML order in the relational data model, and also propose algorithms for translating ordered XPath expressions into SQL using these encoding methods. Finally, we report the results of an experimental study that investigates the performance of the proposed order encoding methods on a workload of ordered XML queries and updates.

1. Introduction

The eXtensible Markup Language (XML) is quickly becoming the *de facto* standard for data exchange over the Internet. The widespread adoption of XML is creating a new set of data management requirements, such as the need to store and query XML documents. Researchers have proposed using relational database systems to satisfy these requirements by devising ways to "shred" (i.e., decompose) XML documents into relations, and translate XML queries into SQL queries over these relations [1][4][7][11][13][14]. However, a key issue with such an approach that has largely been ignored in the research literature is how (and whether) the *ordered* XML data model can be efficiently supported by the unordered relational data model. Supporting XML's ordered data model is crucial for domains like content management, where document data is intrinsically ordered and where queries can exploit this order. For example, if Shakespeare's plays are marked up and stored as XML, the ordering of the acts within a play is relevant, and queries can exploit this order by asking for the second act in a play.

In this paper, we show that XML's ordered data model can indeed be efficiently supported by a relational DBMS. We propose three *order encoding methods* that can be used to represent XML order in the relational data model. These encoding methods are essentially numbering schemes that capture enough information to reconstruct an ordered XML document; that is, they ensure that the mapping from ordered XML to relations is "lossless".

Each encoding method we propose is based on a different approach for achieving a lossless mapping from ordered XML to relations. With the *Global Order* encoding method, the absolute position of each XML element is stored as a data value. With the *Local Order* encoding method, the position of an element relative to its siblings is stored. Finally, the *Dewey Order* encoding method stands as a hybrid of the preceding two methods. These order encoding methods are general and can be used with different approaches for shredding XML documents into relations.

Given these three order encoding methods, the question we would like to answer is: when and why does one encoding method work better than the other? As we shall show, the choice of the order encoding method has a dramatic effect on the performance of ordered XML queries and updates. To answer the above question in a systematic manner, we characterize ordered XML queries (specified in XPath) along three dimensions, and show that each dimension can be supported independently. We then present algorithms for translating XPath queries into SQL using the proposed order encoding methods. Finally, we present an experimental study of the three encoding methods using a workload of ordered XML queries and updates.

Our results show that a relational database system can efficiently support most ordered XPath queries. The best performance is achieved with Global Order for query-mostly workloads, and with Dewey Order for a mix of queries and updates. Our results also show that in some cases machine-translated XPath queries can perform very poorly, requiring manual tuning for optimal performance. However, poor performance in those cases is not due to a flaw in the translation algorithm. Rather, it can be attributed to the fact that the relational database system does not understand the hierarchical structure of XML and the semantics of XPath queries. We discuss these limitations and outline possible solutions.

In summary, this paper presents the first comprehensive study of how XML's ordered data model can be supported using a relational database system.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the ordered XML data model,

* The work was done while the author was visiting IBM Almaden Research Center.

ACM SIGMOD'2002, June 4-6, Madison, Wisconsin, USA.

and also describes the order-based functionality in XML query languages. Section 4 presents three order encoding methods, while Section 5 shows how they can be used with well-known approaches for shredding documents. Section 6 presents an algorithm for translating an XPath expression to SQL. Section 7 describes our experimental results, and, finally, Section 7.6 concludes the paper.

2. Related Work

Research projects such as SilkRoute [5][6] and XPERANTO [2][12] have proposed techniques for efficiently publishing relational data as XML. Commercial database products such as SQL Server, Oracle, and DB2 also provide support for publishing relational data as XML. However, support for ordered XML is not crucial in that application since the underlying relational data is not ordered.

More closely related to this paper is the research on storing and querying XML documents using relational database systems. In that context, there have been many techniques proposed for "shredding" XML documents into relations and for translating XML queries into SQL queries over those relations [1][4][7][11][13][14]. The issues of updating XML data stored in relations [16] and indexing XML data [3][8][24] have also been studied. However, none of these studies provide a comprehensive treatment of XML order. The main goal of this paper is to evaluate the effect of order on all aspects of XML document processing: storage, reconstruction, querying, and updating.

The problem of optimizing queries over ordered relations (sequences) has been studied in the context of sequence database systems [10]. Our work differs from this in two respects. First, instead of relying on an ordered data model, we treat order as a data value. This enables our solutions to be employed in commercial relational database systems optimized for the unordered relational data model. Second, unlike the flat (single-level) ordering considered in sequence database systems, we consider the more general problem of nested XML order.

Temporal databases deal with the ordered time domain, and often treat time as a data value in the underlying database system [15]. Our approach, however, deals with the more general nested ordering in XML documents, and focuses on order encoding methods for efficient evaluation of ordered XML queries.

3. Ordered XML: Data Model, Query Languages and Query Dimensions

In this section, we describe XML's ordered data model and the order-based functionality present in standard XML query languages. We then identify three dimensions of XML order that are key to processing queries over ordered XML documents.

3.1 The XML Data Model

An XML document can be viewed as a tree [19], where leaf nodes correspond to data values (text) and internal nodes correspond to XML elements. Order is a salient feature of the XML data model. Accordingly, an XML document tree is implicitly ordered according to the order of the elements in the XML document. This implicit ordering is referred to as *document order*.

In addition to element and text nodes, an XML document tree can contain attribute nodes. Since attribute nodes are similar to element nodes without subelements, and since XML attributes are not ordered, we do not consider attributes in this study. An XML document can also have a Document Type Descriptor (DTD) [20] associated with it. A DTD provides schema information about XML documents.

3.2 Order in XML Query Languages

In this section, we discuss the order-based functionality in two XML query languages, XPath [21] and XQuery [22]. XPath is a relatively simple language that has been recommended by the World Wide Web Consortium (W3C). XQuery, that is still under W3C development, is a more complex language based on XPath.

3.2.1 Order-based Functionality in XPath

XPath is a language for specifying navigation within an XML document. The result of evaluating an XPath expression on a given XML document is a *set* of nodes sorted according to document order. We will say that the result nodes are *selected* by an XPath expression. An XPath expression has the following syntax: `Path ::= /Step`$_1$`/Step`$_2$`/.../Step`$_N$ where each XPath `Step` is defined as follows:

```
Step ::= Axis :: Node-test Predicate*
```

An XPath expression is evaluated sequentially, "step" by "step". An XPath step is applied to a single node (the *context* node) and selects a set of result nodes. Each node of the result node set is then used as the context node to evaluate the following step. The initial context node is the root of the input document. The result of evaluating an XPath expression is the union of nodesets selected by the last step.

Within an XPath step, `Axis` specifies the "direction" in which the document should be navigated. XPath supports 12 axes for navigation. For example, if `Axis` equals `child`, the step would consider all child nodes of the context node. `Node-test` specifies a simple test on the XML nodes found along the step's axis. The most commonly used `node-test` examines node names. For example, the step `child::title` would select child nodes that are titles. Another `node-test` that is often used is '`*`', which evaluates to true for all element nodes. Hence, `child::*` would select all subelements of the context node.

An XPath step can also include a sequence of predicates. The predicates are applied to the node set selected by the step. Only nodes for which all predicates evaluate to true are returned. Since the focus of our work is on order-based queries, we will limit ourselves to predicates of the form `[position()=n]`. Such a predicate selects all nodes whose position (index) within the context node set equals `n`. The exact meaning of `position` depends on the axis used at the step. For example, `child::title[position()=2]` selects the second child with name "title" whereas `descendant::title [position()=2]` selects the second such a descendant. Our work is easily extended to handle more complex predicates involving `position()`.

In addition to the syntax just described, XPath also supports an abbreviated syntax. For example, the name of the axis can be omitted, in which case it defaults to `child`. Also, if a predicate expression evaluates to an integer value, then that value is considered to be the `position` of the node selected. Hence, the step `title[2]` would select the second title child of the context node. Another commonly used abbreviation is the empty step, //, that selects the context node and all of its descendant nodes. For example, `//title` would select all title nodes anywhere in the input document.

It is important to emphasize that the result of an XPath step (or an entire expression) may not contain duplicates. In addition, the nodes in the result have to be in document order.

3.2.2 Order-based Functionality in XQuery

Since XQuery is based on XPath, the preceding discussion also applies to XQuery. Additionally, XQuery includes `BEFORE` and `AFTER` operators that take two node sequences (XPath expressions) and return the nodes from the first sequence that are before or after some node in the second sequence, respectively. Also, XQuery supports *range* predicates, e.g., `2 TO 5`, which return a contiguous sequence of integers that can be used as a predicate to select a range of elements from a sequence. For example, `/play/act[2 TO 4]` should return the second, third, and forth `acts` in document order.

3.3 Evaluation Modes for XML Queries

Neither the XPath nor the XQuery recommendations [21][22] specify how the nodes selected by query expressions are to be returned to the application. It seems reasonable to assume two possible scenarios (evaluation modes):

Select mode: In this mode, the nodes in an input XML document are assumed to have unique identifiers (IDs). The application needs to identify the nodes selected by an XPath expression. Accordingly, the result of evaluating an XPath expression is an ordered set of node IDs. Given a node ID, the application should be able to extract the XML element corresponding to the node, if needed.

Reconstruct mode: This mode combines selection and extraction. The XML element trees corresponding to the selected nodes are extracted from the input XML document. If the input document is stored in a shredded form, one can say that the elements corresponding to the selected nodes are being *reconstructed*, thus the name. The result of evaluating an XPath expression in reconstruct mode is an ordered set of XML elements.

Clearly, query evaluation in select mode should be simpler than in reconstruct mode. However, preserving document order is essential in either case, as described in the following section.

3.4 The Three Dimensions of XML Order

We characterize the requirements for supporting XML's ordered data model along three dimensions: (a) evaluation of order-based XPath axes and functions, (b) result set ordering (*inter-element* order), and (c) ordered element reconstruction (*intra-element* order). Each of these is discussed in more detail below.

3.4.1 Evaluation of Order-Based Axes And Functions

Most XPath axes, e.g, `child`, can be evaluated even when the input document is stored without order. The following XPath axes, on the other hand, explicitly require document order:

`preceding, following`: These axes select all nodes before (after) the context node *excluding* any descendants (ancestors).

`preceding-sibling, following-sibling`: These axes select all preceding (following) sibling nodes of the context node.

As far as order-based functions go, XPath provides the `position()` function, as described earlier.

3.4.2 Result Set Ordering (Inter-Element Order)

The elements selected by an XML query must be returned in document order. We refer to this requirement as *inter-element order* because it enforces document order among result elements. Inter-element order is important in both XML query evaluation modes. For example, evaluating the XPath expression `/play/act` should return `act` node IDs (in *select* mode) or elements (in *reconstruct* mode) according to document order.

3.4.3 Ordered Element Reconstruction (Intra-Element Order)

In reconstruct mode, data within elements must also be returned in document order. We refer to this requirement as *intra-element order* because it enforces document order within result elements. For example, evaluating the XPath expression `/play/act[2]` in reconstruct mode should return the second `act` with all its data and sub-elements in document order.

4. XML Order Encoding Methods

In order to store and query shredded XML documents using a relational database system, we need some mechanism to capture document order in the relational data model. This is accomplished by encoding each node's position in an XML document as a data value. A variety of order encoding methods are possible, but a valid order encoding method *must* allow for reconstruction of the *original* ordered XML document. In additional, an order encoding method should allow for translation of ordered XML queries and updates into *efficient* SQL.

Note that the first requirement ensures a "lossless" mapping from XML to relations, while the second requirement is desired for performance reasons. Unfortunately, as we will show, there appears to be no single encoding method that is simultaneously optimal for queries and updates. Encoding methods that perform well on queries tend to incur more overhead on updates.

In light of the above discussion, we focus our attention on three "lossless" order encoding methods that span the spectrum of query and update performance. The first of these encoding methods, *Global Order*, performs the best on queries. At the other end of the spectrum is *Local Order*, which performs best on updates. Finally, *Dewey Order* is a hybrid of the preceding two methods, which performs reasonably well on both queries and updates. The order encoding methods that we describe are general and can be applied to different approaches for shredding XML. In Section 5, we will illustrate this using two different approaches for shredding XML

4.1 Global Order Encoding

With Global Order, each node is assigned a number that represents the node's absolute position in the document. For example, an element's position can be encoded as the byte offset of its opening tag from the beginning of the document. Note that a node's position need not have an actual meaning such as a byte offset. Any numbering scheme can be used as long as it is consistent with document order. See Figure 1 for an illustration of Global Order.

Global Order makes it easy to answer XPath queries that use order axes, such as `following` and `following-sibling`. Indeed, such queries can be translated into simple comparison conditions between node positions. (We will describe the algorithm for this translation in more detail in Section 6.) Also, Global Order makes it easy to handle both intra-element and inter-element ordering requirements because the global document order is readily available in the encoded values.

If updates are to be supported with Global Order and other order encoding methods, performance can be improved using *sparse* numbering. With sparse numbering, deletion of an XML fragment does not require that remaining nodes be renumbered.

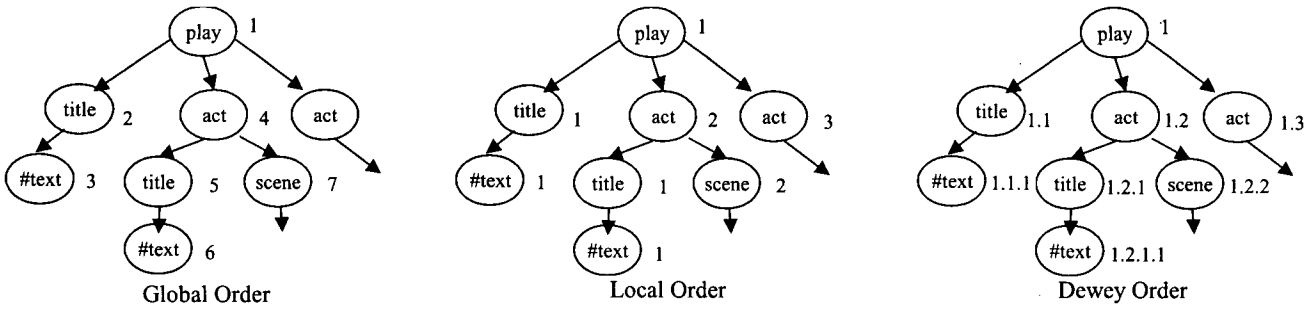

Figure 1. Illustration of Order Encoding Methods

Additionally, gaps are left between assigned position values when the initial numbering is performed. As a result, insertions may not require renumbering to accommodate a new XML fragment. In the worst case, however, the number of available position values may be smaller than the number of nodes in the XML fragment being inserted. In this case, some (or all) of the elements *following* the newly inserted fragment will have to be renumbered, see Figure 2.

It may seem beneficial to use real (floating-point) values instead of integers to represent a position. In theory, there is an infinite number of real values between any pair of values, so insertions would never require renumbering. But in reality, both real and integer values are represented with the same number of bits. As a result, there is a finite number of values between any two real values stored in the computer and using real values instead of integers does not provide any benefit.

Poor insertion performance is thus a potential weakness of Global Order, which we confirm experimentally in Section 7. We now describe another order encoding method that can handle insertions more efficiently.

4.2 Local (Sibling) Order Encoding

With Local Order, each node is assigned a number that represents its relative position among its siblings (see Figure 1). To see that Local Order is sufficient to recreate document order, note that combining a node's position with that of its ancestors yields a path vector that uniquely identifies the absolute position of the node within the document. In other words, such a path vector provides global node ordering.

As shown in Figure 2, the advantage of Local Order is the low overhead incurred by updates. Only the *following siblings* of the new node may need to be renumbered. As with Global Order, sparse numbering can improve performance of updates. However, the low overhead of insertions comes at the cost of evaluating ordered XML queries. With Local Order, order axes like `following` and `preceding` are difficult to evaluate since no global order information is available. In the next section, we describe a hybrid scheme that strikes a balance between the advantages and weaknesses of Global and Local Order.

4.3 Dewey Order Encoding

Dewey Order is based on Dewey Decimal Classification developed for general knowledge classification [9]. With Dewey Order, each node is assigned a vector that represents the path from the document's root to the node. Each component of the path represents the local order of an ancestor node, as illustrated in Figure 1. Dewey Order is "lossless" because each path uniquely identifies the absolute position of the node within the document.

Since Dewey paths provide global node ordering, query processing in Dewey Order is similar to that in Global Order. In terms of overhead incurred by updates, Dewey Order represents the middle ground between Global and Local Order. Only the following *siblings and their descendants* may need to be renumbered, as shown in Figure 2.

Even though Dewey Order combines many of the advantages of Global Order and Sibling Order, one of its potential disadvantages is the extra space required to store paths from the root to each node. In Section 6.2.1, we will describe a UTF-8 [23] based representation that can help minimize this overhead.

4.4 Same-Sibling (Partial) Order Encoding

It may seem possible to use the following variation of Local Order. Instead of assigning each node a number according to its position among siblings, only the siblings with the same tag name are considered. We refer to this method as *Same-Sibling Order*. Unfortunately, Same-Sibling Order alone is not sufficient to recreate document order because it does not define a *total* order between all siblings (only partial order is defined). Hence, Same-Sibling Order can only be used in conjunction with an order encoding method that defines a total order among nodes, such as one of the three described above. Despite its shortcomings, Same-Sibling Order can be beneficial on queries that select the *n-th* sub-element with a certain tag name (e.g., `/play/act[2]`).

5. Shredding Ordered XML into Relations

In this section, we describe how our three order encoding methods can be used with well-known approaches for shredding XML documents into relations. We consider two cases: when the schema of input documents is unknown (the *schema-less* case), and when the schema is known (the *schema-aware* case).

5.1 The Schema-less Case

In many applications, the schema of input documents is unknown. The Edge shredding approach has been proposed recently to handle this case [7]. A single relation, the *Edge table*, is used to store an entire document. When preserving document order is not an issue, the Edge table is defined as follows:

```
Edge(id, parent_id, name, value)
```

Each Edge tuple represents a node in the XML document tree. The `id` column corresponds to the node's ID and also serves as the primary key of the relation. The `parent_id` column provides a "link" (i.e., foreign key) to the node's parent. The `name` column is used to store the tag name of element nodes, the `value` column is used for text values of text nodes.

Instead of the tag name, the path from the root to an element node can be stored, e.g., `/play/act`. In order to reduce storage requirements, a separate relation (the *Path table*) can be used to store paths and their identifiers [14][25]. The `name` column of `Edge` can then be replaced with a `path_id` column. A Path

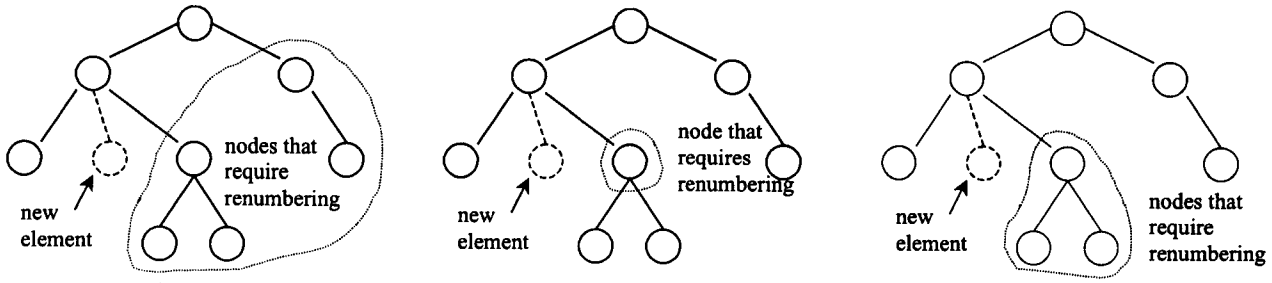

Figure 2. The worst case renumbering scenarios for Global, Local, and Dewey order encodings.

table will typically be small since it only records unique paths rather than all path instances.

Storing paths instead of tag names can greatly simplify query processing. For example, to find all nodes that satisfy the XPath expression `/play/act/scene`, a single join of the Edge and Path tables can be used to identify the result tuples. Without a Path table, 3 self-joins of the Edge table would be required.

5.1.1 Storing Order Information

When document order needs to be preserved, order information must be stored along with document structure and data. Depending on the order encoding method used, the basic Edge approach has to be adapted as follows:

Global Order: If node IDs are assigned according to document order, then document order will be preserved. As will be seen shortly, it is also beneficial to add a column to store the ID of the last descendant of a node (`end_desc_id`). The Edge table can then be defined as follows:

```
Edge(id, parent_id, end_desc_id, path_id, value)
```

Local Order: Since the relative position of a node among its siblings does not uniquely identify a node in a document, unique node IDs still need to be assigned (that do not have to follow document order). A new column needs to be added to represent the position of a node among its siblings (the sibling index of a node, `sIndex`):

```
Edge(id, parent_id, sIndex, path_id, value)
```

Dewey Order: A Dewey path represents both order and ancestor information. As a result, with Dewey Order, Edge is especially simple: `Edge(dewey, path_id, value)`.

Since the length of a node's Dewey path is unknown a priori, the `dewey` column has to be stored as a variable-length byte string.

5.2 The Schema-aware Case

When an XML Schema (or DTD) is available, more efficient shredding techniques based on *Inlining* can be used [11]. With Inlining, child elements that can occur at most once are stored as one or more columns in the same relation as their parent. A new relation is created only for child elements that can occur multiple times. For example, suppose a schema specifies that a `play` can have at most one `title`, but can have an arbitrary number of `acts`. In that case, `title` would be inlined as a column in the relation for `play`, while `acts` would be stored in a separate relation that is linked to the `play` relation using a foreign key.

One advantage of Inlining is the possibility of more efficient navigation from an element to its subelements. For example, in the Edge approach, retrieving a subelement of a given element requires a self-join of the Edge relation. With Inlining, the subelement may be immediately available if the subelement is stored in the same tuple as the parent element. As a consequence, reconstruction of a stored XML element is more efficient as well. Another advantage of Inlining is physical partitioning of data. Instead of storing entire documents in a single Edge table, Inlining shreds XML documents into a set of tables according to the document schema. As a result, queries tend to access less data, which leads to better performance (see Section 7).

It is straightforward to adapt Inlining to work with Global, Local, or Dewey Order. As with Edge shredding, an additional column is added to every relation to encode document order. Note that there is no need to have a separate column for storing the order information of inlined elements, since the position of such elements can be determined from the position of their parent element and the document schema.

6. Translating Ordered XML Queries and Updates into SQL

We now describe algorithms for translating ordered XML queries and updates into SQL queries based on the three order encoding methods discussed earlier. We will start with the algorithm for Global Order. Since essentially the same algorithm can be used for both Edge and Inlining shredding, we focus on Edge shredding and then outline how it can be extended for Inlining.

6.1 Query Translation for Global Order

The algorithm for translating an ordered XPath query into SQL is shown in Figure 3. (We will discuss extensions for ordered XQuery functions and updates shortly.) We first describe the algorithm at a high level, and then provide additional details. In the interest of space, we mainly focus on the translation of ordered axes, omitting the details of how a sequence of unordered child steps can be combined into a single predicate on the *Path table* [14][25].

As shown, the algorithm in Figure 3 initially generates the SQL fragment to select the root elements of the stored XML documents (lines 4-6). Then, using the root elements as the initial context nodes, the algorithm generates the SQL fragments for each "step" of the XPath query being translated in order to produce new context nodes (line 8). The context nodes produced by the last step constitute the query result (lines 10-11).

The SQL fragments generated by the algorithm for the root nodes and for each step are represented using "with clauses" (also referred to as inlined views). We use "with clauses" instead of nested sub-queries because they can also be used for defining recursion, which will be used in the query translation for Local Order.

We now present the details of translating each step in an XPath query. An XPath step is translated to SQL by generating and concatenating SQL fragments for the axis of the step, the node test of the step, and the predicates associated with the step (line15). As an example, consider the generation of the SQL

fragment for XPath's child axis. Let us assume that the current context node set is represented by the "inlined view" Q_q that returned the ids and end_descendant_ids of the context nodes. To evaluate the child axis, we need to determine the child nodes of the context nodes. This task can be performed by joining the parent_id column of the Edge table with the id column of the context nodes (lines19-21).

So far, we have focused on the translation of general XPath features; we now turn our attention to ordered axes and functions.

6.1.1 *Translation of* following

Recall that in order to evaluate the following axis, we need to determine all the nodes that occur after the context node in document order (excluding the descendants of the context node). This is easy to do with Global Order; we just need to select those nodes in the Edge table whose id value is greater than the end_descendant_id value of the context node (lines 22-24). The preceding axis can be evaluated similarly.

6.1.2 *Translation of* following-sibling

To evaluate the following-sibling axis, we need to determine all the sibling nodes of the context node that occur after the context node. We can achieve solve this problem by selecting the nodes in the Edge table that (a) have an id value that is greater than id value of the context node, and (b) have the same parent_id value the context node (lines 25-27). The preceding-sibling axis can be evaluated similarly.

6.1.3 *Translation of Position-based Predicates*

Consider the following XPath expression involving a position based predicate: /play/act/scene[2]. A naïve way of translating this into SQL is to use a subquery and the "count" function to find the number of scene nodes occurring before the context scene node, and then select only those nodes for which the subquery returns one. Clearly, this approach is likely to be very inefficient since the subquery is evaluated for each context scene node. Fortunately, SQL'99 has an efficient "rank" function, originally proposed as an OLAP extension. We now show that the "rank" function can be used to evaluate position-based predicates in XML queries (the performance advantages of the rank approach are studied in Section 7).

Lines 51-56 in Figure 3 show the SQL fragment to translate position-based predicates using rank. Essentially, the rank clause is applied to the context nodes to give them a position (lines 32-37), and only those context nodes that satisfy the position-based predicate are selected (line 37). The rank clause has two parts. The PARTITION BY part is applied to the context nodes to partition them based on the parent_id (thus, sibling scenes in /act/play/scene[2] will be in the same partition). Within each partition, the ORDER BY part is used to map a sparse ordering into a dense one (thus, all sibling scenes will be ordered starting from one: 1, 2, ... and so on). Given the dense numbering, the desired position can then be selected (in our example, T.pos = 2 on line 37).

If position-based predicates are used often and updates are rare, it is beneficial to add an additional column to store *dense* same-sibling (ssIndex) order information (see Section 4.4). As a result, ranking nodes would be unnecessary and a simple selection could be used. Note, however, with *dense* ordering every update (insert or delete) has to be followed by node renumbering.

6.1.4 *Translation of XQuery Operators*

The BEFORE and AFTER operators of XQuery are similar to preceding and following axes of XPath, and can be translated similarly. The range selection operator, e.g., [2 to 5], can be handled similarly to position-based predicates, except that the final selection condition will be a range predicate instead of an equality predicate.

6.1.5 *Enforcing Inter- and Intra-Element Order*

In order to ensure that the results of an XPath expression are in document order (*inter*-element order), we can simply sort the result nodes on Global Order ID. Reconstructing the content of these nodes, including that of their descendants, in document order (*intra*-element order) is slightly more complex. First, we identify all of the descendants of the selected nodes by retrieving the nodes whose id value lies between the id and end_descendant_id of a context node (line 16). We then sort all of these nodes based on the ids of the context node *and* the corresponding descendant node (line 17).

Note that in general, we need to sort by the id of the context node in addition to the id of the descendant nodes. This is because the context nodes could have an overlapping set of descendants. For example, consider the XPath expression //speech. This could return a set of context nodes such that speech s1 could be a descendant of another speech s2. To establish intra-element document order, we need to ensure that all of s1's descendants immediately follow s1, and similarly all of s2's descendants immediately follow s2. Merely sorting on the id of the descendants of s1 and s2 will result in an interleaving of their descendants. This problem is avoided by including the id of the context node as an extra sort field. Note, however, that if the XPath expression is such that there is no chance of overlapping descendants, e.g., /act/play/scene/speech, then, sorting on the id of the descendant nodes only is sufficient.

For the purposes of this study, we do not consider adding XML tags to the ordered results (i.e., our goal is to return a stream of tuples in document order). A constant space tagger such as the one proposed in [12] can be used to add XML tags to the output data on-the-fly.

6.1.6 *Translating XML Updates*

Since XPath and XQuery do not yet provide support for updates, we use the update extensions proposed in [16]. We consider only inserts and deletions because these operations affect order (specifically, we do not consider modifications to XML data values). Insertions are specified using two parts. The first part is an XPath expression that selects a set of context nodes, and the second part is the XML element to be inserted after the selected context nodes. Deletions are specified as an XPath expression, whereby all the selected nodes are deleted.

It is easy to see that deletions can be handled by evaluating the associated XPath expression (as in the case of queries), and then deleting the selected nodes. In particular, there is no need for renumbering if sparse numbering is used. Insertions, on the other hand, are subtler because the nodes following the inserted XML may need to be renumbered in order to maintain the global ordering (see Section 4 for more details). In this context, two general methods of executing inserts seem feasible. In the first *conservative* method, a query is first executed to find out if there is a conflict and renumbering is necessary (note that this query can be combined with the XPath query for selecting the context

```
1  q = 1; // global WITH subquery count
2  func translateXPath(xPath) {
3    // retrieve root node info
4    sql =      "WITH Q1 (id, parent_id, end_descendant_id) AS (
5               SELECT id, parent_id, end_descendant_id
6               FROM Edge WHERE parent_id = -1);
7    // translate each step
8    for (i=1; i <= length(xPath); i++) { sql += translateStep(xPath[i]); }
9    // assume result elements need to be reconstructed
10   sql +=     "SELECT E.* FROM Edge E, Qq as Q WHERE E.id >= Q.id AND E.id <= Q.end_descendant_id
11              ORDER BY Q.id, E.id;
12   return sql;
13 }
14 func translateStep (step) {
15  return translateAxis(step.axis) + translateNodeTest(step.nodeTest)+ translatePredicates(step);
16 }
17 func translateAxis (axis) {
18  switch kind(axis) {
19  CHILD:               sql = "WITH Qq+1 (id, parent_id, end_descendant_id) AS (
20                               SELECT E.id, E.parent_id, E.end_descendant_id
21                               FROM Edge E, Qq as Q WHERE E.parent_id = Q.id)";
22  FOLLOWING:           sql = "WITH Qq+1 (id, parent_id, end_descendant_id) AS (
23                               SELECT E.id, E.parent_id, E.end_descendant_id
24                               FROM Edge E, Qq as Q WHERE E.id > Q.end_descendant_id)";
25  FOLLOWING-SIBLING:   sql = "WITH Qq+1 (id, parent_id, end_descendant_id) AS (
26                              SELECT E.id, E.parent_id, E.end_descendant_id
27                              FROM Edge E, Qq as Q WHERE E.id> Q.id AND E.parent_id = Q.parent_id)";
28  ...
29 }
30 func translatePredicate (predicate) {
31  switch kind(predicate) {
32  [n]: sql ="WITH Qq+1 (id, parent_id, end_descendant_id) AS (
33             SELECT T.id, T.parent_id, T.end_descendant_id
34             FROM (SELECT Q.id, Q.parent_id, Q.end_descendant_id,
35                          RANK OVER (PARTITION BY Q.parent_id ORDER BY Q.id) pos
36                   FROM Qq Q) as T
37             WHERE T.pos = n)";
38    ...
39  q++; return sql;
40 }
```

Figure 3. The XPath-toSQL Transaltion Algorithm for Edge with Global Order.

nodes). If there is a conflict, the nodes following the context nodes are renumbered.

The second method is more *optimistic*. It can be used only with dense numbering or when a single node is being inserted. Given that global order information is unique for each node, a constraint can be defined in the relational database system. As a result, a query to detect the need for renumbering is not necessary, and a new XML node can be inserted right away. If the relational database system detects a duplicate order value, an error is reported. At this point, the algorithm backs down to the conservative method: renumbering is performed and the new data is inserted. Clearly, if conflicts are rare the optimistic approach should be advantageous. The probability of a conflict will depend, however, on the algorithm that is used for sparse numbering and the insert pattern. We evaluate the relative performance of the conservative and optimistic strategies in Section 7.

6.2 Query Translation for Dewey Order

We now describe the translation algorithm for Dewey Order. The key insight here is that Dewey Order can essentially be treated as a Global Order (and use the same translation algorithm) if we can enforce certain properties on Dewey paths. Also, since Dewey Order encodes parent and descendant information implicitly in the Dewey path, there is no need to explicitly store these values as in the case of Global Order; rather these values can be derived from the Dewey path. We now discuss these two issues in more detail.

6.2.1 Enforcing Global Order using Dewey Paths

In order to enforce global (document) ordering using Dewey paths, we need to ensure that the result of a comparison between two Dewey paths is consistent with document order. This will ensure that Dewey Order can be used just like Global Order in "order by" and "comparison" operators in SQL. It turns out, however, that a simple concatenation of the components of the Dewey path using a separator such as "." does not work. Indeed, consider two Dewey paths "1.2" and "10.2". A comparison of these Dewey paths will indicate that "10.2" occurs before "1.2" (because '0' occurs before '.'). Clearly, this answer is not consistent with document order.

The above problem can be overcome by allocating a *fixed number of bytes* for each component of a Dewey path. In our example, the paths would be represented as "00001.00002" and "00010.00002". This simple approach, however, may result in too much storage overhead because the number of bytes allocated for each component is dependent on the largest number of sub-elements that can occur under an element. If each element has a small number of sub-elements on average, storage requirements will be unjustifiably high.

In order to address this issue, we use UTF-8 encoding [23] as an efficient way to represent Dewey paths. In UTF-8, a variable number of bytes are used to encode different integer values. Smaller values use a smaller number of bytes. For example, if the

value is smaller than 2^7=128, it is encoded with a single byte 0xxxxxxx where x represents a bit used for value encoding. The values between 2^7 and 2^{11} are encoded with two bytes 110xxxxx 10xxxxxx, and so on. To represent an entire Dewey path with UTF-8, each component of the path is encoded in UTF-8 and then concatenated. This enables two Dewey paths to be compared as simple byte strings, without incurring a large space overhead.

6.2.2 Inferring parent_id in Dewey

Recall that in the query translation algorithm for Global Order, we used the `parent_id` and `end_descendant_id` values that were stored explicitly. Dewey Order eliminates the need to explicitly store these values because they are implicitly encoded in a Dewey path. Specifically, the `parent_id` of a node with Dewey path `dp` is just `prefix(dp)`, where `prefix(dp)` is a function that removes the last Dewey component of `dp`. Similarly, the `end_descendant_id` of a node with Dewey path `dp` can be calculated from the following expression: `dp || x'FF'`. This expression appends an "all-ones" byte, `x'FF'`, as the last component of the new Dewey path, which represents the maximum possible descendant `id`. (Note, `x'FF'` cannot be used in a valid UTF-8 encoding.) We have implemented `prefix()` as a user-defined function in the relational database system. This function, along with the UTF-8 encoding, allows using the Global Order query translation algorithm for Dewey Order.

The conservative and optimistic insertion algorithms described for Global Order also apply to Dewey Order. However, in case of conflicts, fewer nodes need to be renumbered in the Dewey Order case (see Section 4 for more details).

6.3 Query Translation for Local Order

The translation algorithm for Local Order is different from the one for Global Order and Dewey Order because of the lack of global order information. Some features, however, share the same translation. We will describe the common features first, and then illustrate the key differences between the algorithms.

Order axes like `following-sibling` and `preceding-sibling` and ordered functions like `[n]` and `[m TO n]` can be the handled in the same way as Global Order and Dewey Order. Indeed, the evaluation of these axes and functions only requires an ordering among the sibling nodes.

The more challenging aspect of the translation using Local Order is for "global" axes and operators, such as `following`, `preceding`, `BEFORE` and `AFTER`. Because of the local scope of the numbering, these primitives cannot be directly implemented using a comparison of order values. Instead, the following technique can be used. Consider the evaluation of the `following` axis on a context node. We first determine all the ancestors of the context node. Let us call the set of ancestors `anc`. We then compute all of `anc`'s `following-siblings`, as described above, to get `anc_sib`. Finally, we compute the descendants of `anc_sib` to get the desired result.

The main challenge of this technique lies in computing the ancestors and descendants. Consider the problem of computing the ancestors of a context node. With fully specified XPaths, like `/play/act/scene`, this problem is easy. However, with more complex XPath expressions, such as `//speech`, and without knowledge of the XML schema, the problem is much harder since it is not clear how deep the context node is in the XML hierarchy. Therefore, we need to employ recursion to compute all of its ancestors. Similarly, computing descendants also needs recursion. We use SQL's least fix-point recursion (using "with clauses") to compute ancestors and descendants. It is easy to extend the above algorithm for `following` to other related primitives such as `preceding`, `BEFORE` and `AFTER`.

We now turn our attention to *inter*-element order. With only local order information available, a global numbering has to be computed to ensure inter-element order. We do this by concatenating the local order information along the path from the root to the context node, essentially building a Dewey path "on the fly". The computed Dewey paths can then be used in a SQL "order by" clause to ensure inter-element order. Computing the Dewey paths can require SQL recursion if the depth of the context nodes is unknown.

Note that *intra*-element order always requires recursion to compute the descendants of context nodes. In addition, Dewey paths need to be computed in order to sort the descendant nodes in document order. Although query evaluation is more complex with Local Order, updates are simpler because fewer nodes need to be renumbered (see Section 4). Both conservative and optimistic insertion methods can be used with Local Order.

6.4 Query Translation in Inlining

So far, we have focused on the Edge shredding approach in describing our translation algorithm. Essentially the same algorithm can be used for the Inlining shredding approach, but with two key extensions. First, in the Inlining method, the XML data can be spread across multiple tables. Therefore, in evaluating ordered axes like `following`, `following-sibling`, `AFTER`, etc., we may need to access a set of tables (instead of just accessing the Edge table). This set of tables is determined using the XML document schema. Similarly, when performing ordered XML element reconstruction, data may need to be accessed from multiple descendant tables. In this case, we union these results together and later order them using the efficient "sorted outer-union" method proposed in the literature [12].

The second difference in the translation algorithm for Inlining is regarding Local Order. Unlike the Edge case, selecting the set of ancestors, descendants, and computing Dewey-like paths "on the fly" does *not* require recursion (unless the schema is itself recursive). This is because the XML schema gives sufficient information about the depth and position of nodes in the XML tree; consequently, regular (non-recursive) joins are sufficient.

One disadvantage of Inlining is that an optimistic insertion algorithm cannot be used easily. Since data is partitioned across multiple tables, a simple "unique" constraint such as the one used to detect order conflicts in a single Edge table, cannot be defined.

7. Experimental Results

We now describe the results of a comprehensive experimental study designed to evaluate and compare the performance of the three order encoding techniques proposed in this paper. For the experiments, we used the freely available Shakespeare's plays dataset [18]. We chose this dataset because it represents a realistic scenario where XML document order is important. Also, this dataset includes document schema, which allowed us to experiment with both the Edge and Inlining shredding approaches. The dataset contains 37 documents, conforming to a single schema [18]. In our experiments, we scaled up this dataset by 5x and 10x to create a more sizeable data workload. Most of our experiments are presented using the 5x dataset, but we also illustrate the scalability properties of our approaches using the 10x dataset (which contains 100MB of ordered XML).

Table 1: Storage reqs. of the 3 encoding methods (5x dataset)

Order scheme	Edge (898,445 tuples)		Inlining (888,900 tuples)	
	Table size	Index size	Table size	Index size
Global	52.1 MB	57.9 MB	44.1 MB	28.9 MB
Local	52.1 MB	87.9 MB	47.7 MB	36.8 MB
Dewey	48.9 MB	38.7 MB	44.5 MB	15.8 MB

All experiments were run on an 866MHz Pentium III processor with 1GB of physical memory running Windows 2000. We used IBM DB2 Version 7.1 for our experiments. For most of our experiments, the database buffer and sort area were both set to 200 MB, which was enough for the entire data set to fit in memory. (We used a smaller, 10M buffer in Section 7.7). In order to ensure consistency, each test query was executed 6 times with the performance results of the first run discarded. A 95% confidence interval was computed to estimate error. Since the estimated error was small (<10%), confidence intervals are not shown in the graphs.

7.1 Storage Requirements

The storage requirements for the three order encoding methods using both Edge and Inlining shredding are shown in Table 1. The two shredding approaches produce almost the same number of tuples because most elements occur multiple times within their parent element (according to the DTD), which makes the elements non-inlinable. The set of indexes that we created was determined through careful analysis of the query plans generated by DB2's optimizer. The size of the path table (for the Edge case) is not shown since the table has only 57 tuples, which equals the number of unique paths in the input documents.

7.2 Test Queries

The test queries that we used are shown in Table 2. We chose these queries for the following reasons. Q1 evaluates reconstruction performance on the entire document set. Q2 evaluates recursive (//) XPath queries. Q3 is very similar to Q2 but does not use recursion. The rest of the queries were chosen to test key aspects of order-based functionality in XPath and XQuery. The test queries were translated to SQL using the algorithm described in Section 6.

As discussed earlier, an XPath query can be executed in two modes. In *select* mode, the result contains only the IDs of the nodes satisfying the XPath expression. In *reconstruct* mode, nodes satisfying the XPath expression are identified and then their corresponding elements are reconstructed from database tuples. Since one of the goals of this paper is to study the overhead of preserving document order, we consider both ordered and unordered versions of the select mode. On the other hand, since most applications will probably expect order within an element to be preserved, we consider only the ordered reconstruction mode. We now turn our attention to the performance results.

7.3 Unordered Selection

XPath requires the result of a query to be in document order. However, it is still useful to know the cost of evaluating unordered XPath expressions in order to infer the overhead of preserving document order. Figure 4 (left) shows the performance of the three order encoding methods for unordered selection, using Edge shredding. Figure 5 (left) shows the corresponding results for Inlining. Clearly, Global Order results in the best performance on all queries, except Q5. Note that Q5 is a rather unusual query, which selects the second child node of a `scene` no matter what the name of the child node is. Such a query can be evaluated efficiently if (dense) local order information is available (as with Local and Dewey Order). However, since Global Order does not have this information, it performs relatively poorly.

The Edge results for Q7 and Q9 show that Local Order is a poor choice for queries that use "global" axes and functions. Recall that Q7 selects `speeches` *following* the second `act`, whereas Q9 selects `speeches` *before* the second `act`. Both queries require global order information. Recreating global order from local order requires recursion (see Section 6.3), which is why Local Order performs poorly on these two queries. Interestingly, Q7 and Q9 perform rather well with Local Order and Inlining. This rather surprising result can be explained as follows. Under Inlining, the schema of the dataset is available to the query translator. The schema information can be used to determine the set of tables with the nodes that can occur before/after the context nodes. As a result, translation into more efficient SQL queries is possible.

As shown, the higher cost of comparing Dewey paths makes it slightly slower than Global Order. On Q8, however, Dewey Order is significantly slower. This query requires a join on `parent_id` (to find following *siblings*). Recall that with Dewey Order, `parent_id` is determined by applying the `prefix()` user-defined function on the `id` of a child (see Section 6.2.2). Thus, the join on `parent_id` is of the form: `prefix(d1) = prefix(d2)`. Because of the use of two user-defined functions, this comparison significantly limits the SQL optimizer's options. A simple way to improve Dewey Order on such queries is to add a column recording the Dewey path of a node's parent, although this modification would lead to higher storage requirements.

7.4 Ordered Selection

Recall that with ordered selection, SQL queries return a set of node IDs in document order. The middle charts in Figure 4 and Figure 5 show the results of our ordered selection experiments. Comparing the results for ordered selection with those for unordered selection, we can ascertain the overhead of preserving

Table 2: Test queries.

Query	Query definition	Node count	Edge tuple count	Inlining tuple count
Q1	`/play`	185	898,445	733,760
Q2	`/play/act//speech`	154,755	856,880	699,035
Q3	`/play/act/scene/speech`	154,665	853,850	696,095
Q4	`/play/act/scene/speech[2]`	3,650	20,620	16,910
Q5	`/play/act/scene/*[2]`	3,740	3,760	3,755
Q6	`/play/act/scene/speech[1 TO 3]`	11,030	76,570	65,375
Q7	`/play/act[2]/following::speech`	91,900	506,615	412,775
Q8	`/play/act/scene/speech/speaker/following-sibling::line[2]`	85,445	85,445	85,445
Q9	`//act/scene/speech BEFORE /play/act[2]`	30,215	172,370	141,680

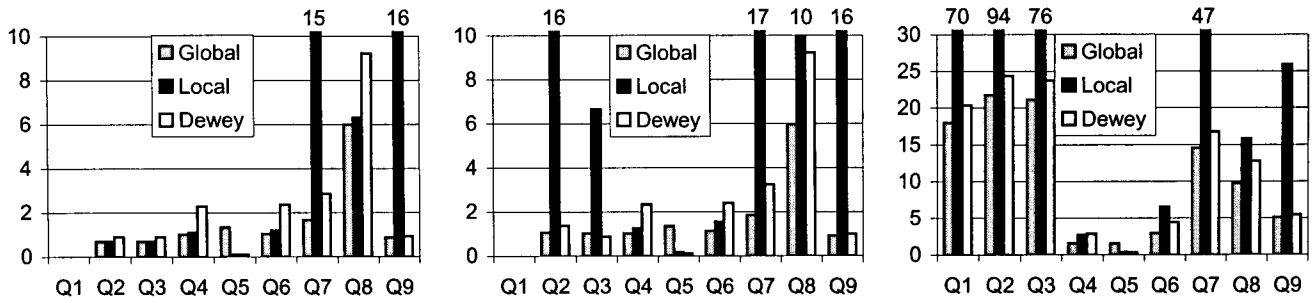

Figure 4. Edge results for unordered selection (left), ordered selection (middle), and reconstruction (right). 5x dataset.

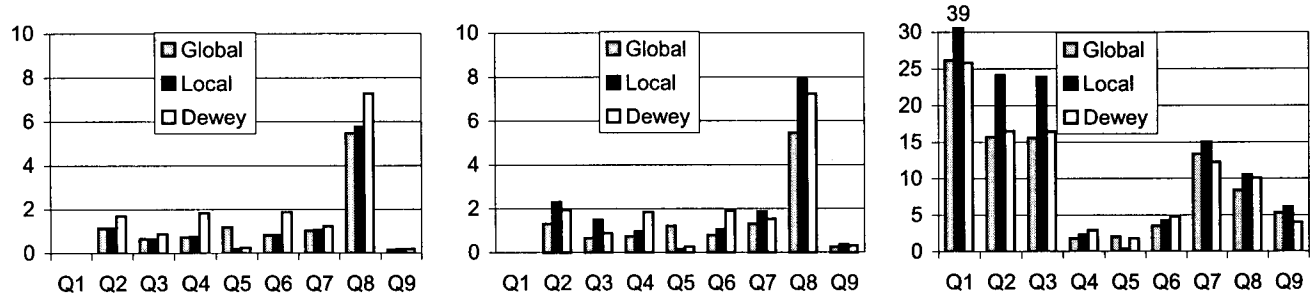

Figure 5. Inlining results for unordered selection (left), ordered selection (middle) and reconstruction (right). 5x dataset.

document order. With Global and Dewey Order, this overhead is quite small. Indeed, in these methods, sorting on the node IDs is sufficient to return results in document order.

With Local Order, however, global ordering has to be generated "on the fly" which, in the case of Edge, requires recursion (see Section 6.3). As a result, the performance of Local Order with Edge is rather poor. Note, however, that Local Order performs reasonably well with Inlining. With Inlining, the set of tables that need to be joined to recreate global order can be determined from the query and the XML schema. As a result, recursion is not necessary (see Section 6.4).

7.5 Reconstruction

In reconstruct mode, entire XML fragments need to be extracted from the database in document order. Our initial results with Global Order and Edge were extremely poor because the DB2 optimizer does not understand containment queries well. As a result, the optimizer picked poor execution plans for Global Order and Edge, and we had to "tune" our SQL queries to make it choose better plans. More will be said about this problem shortly. For the results in Figure 4 (right), we used tuned SQL queries, since they are a better measure of Global Order's inherent performance.

The results shown in the charts on the right of Figure 4 and Figure 5 indicate that Global Order results in the best performance on all queries except Q5. Again, Dewey Order is slightly slower than Global Order due to the higher cost of comparing Dewey paths. Local Order results in noticeably lower performance, especially with Edge, which is caused by use of recursion. Local Order with Inlining does not require recursion (Section 6.4), but still suffers the high overhead of generating a global order on the fly (through sorting on a set of columns instead of a single column).

7.5.1 Performance Problems with Global Order and Edge

Our initial reconstruction results with Global Order and Edge were simply disastrous, as shown in Figure 6. In this set of experiments, we used the original 1x dataset *without scaling* because of the large running times. As shown, five out of nine test queries initially took more than forty seconds to complete. Q3 was especially slow, taking 35 minutes to complete.

An interesting thing to note, however, is that even though Q2 and Q3 are similar, Q3 was initially 400 times slower than Q2! If anything, Q2 should have been slower than Q3 because it is based on a more complex XPath expression that includes "//"? The crucial difference here lies in the SQL generated for Q2 and Q3. In particular, since Q2 has potentially overlapping descendants, it has to be ordered by the id of the context nodes in addition to the id of the descendants (see Section 6.1.5). On the other hand, Q3 cannot have overlapping descendants, and hence is ordered only by the id of its descendants.

The query plans for Q2 and Q3 reveal why Q3 ran so much slower than Q2 (see Figure 7 and Figure 8). The plan for Q2 joins the Path and Edge tables to find `speech` tuples. Then Edge tuples that are contained in each `speech` are retrieved. Finally, the result is sorted on (`Q.speech_id, E.id`) to group the subelements contained in each `speech`. The plan for Q3, on the other hand, avoids sorting by scanning the entire Edge table through an index (on `E.id`) and checking every tuple for containment within a `speech`.

Either plan can be used for Q3. So why did the optimizer choose the wrong plan for Q3? The DB2 optimizer based its decision on the cardinality estimate of the following containment predicate:

```
E.id ≥ Q.speech_id AND E.id ≤ Q.speech_end_desc_id
```

The optimizer grossly overestimated the cardinality of this predicate and consequently, the amount of data that needed to be sorted. To avoid what it thought was an expensive sort, the optimizer chose the plan in Figure 8 for Q3 when it should have chosen the plan in Figure 7.

Our "solution" to this problem was to trick the DB2 optimizer into picking the plan in Figure 7 for Q3. This was accomplished by replacing the single `E.id` column in the ORDER BY clause of

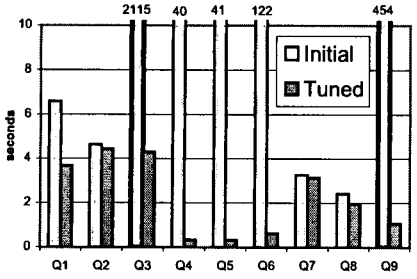

Figure 6. Intial and "tuned" reconstruction performance of Global Order with Edge, 1x dataset.

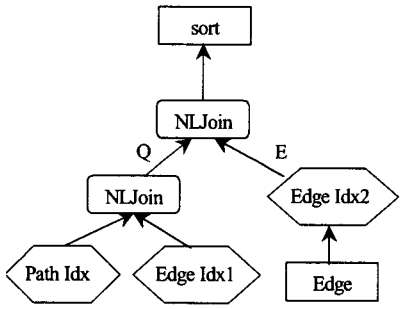

Figure 7. The initial query plan Q2.

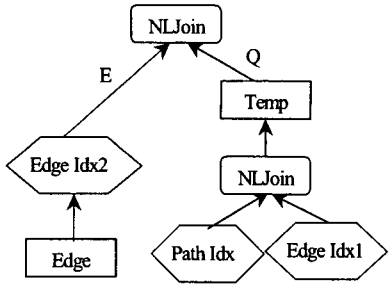

Figure 8. The initial query plan for Q3.

Q3 with (`Q.speech_id, E.id`). This forced the sort to be done after the final join, which eliminated the plan in Figure 8 as an option. Clearly, the extra ORDER BY column does not affect tuple ordering. This "tuning" trick was also used on other queries that "confused" the optimizer.

A more general solution is, of course, needed. A recent study [24] concluded that a multi-predicate merge join should be used to evaluate containment predicates like the one above. The problem is that, without knowing the hierarchical structure of the data, it is impossible to produce a reasonable cardinality estimate for containment predicates. As a result, the optimizer will not be able to decide when to use a multi-predicate merge join. One could argue that a multi-predicate merge join should *always* be used to evaluate containment predicates. However, it is easy to construct examples where a multi-predicate merge join performs worse than a nested-loop join on containment predicates.

Interestingly, we did not experience the performance problems of Global Order with Dewey Order. It turns out there is no "deep" reason for this. The containment predicate with Dewey Order is somewhat different than with Global Order. Specifically, Dewey Order requires the following containment predicate: `E.dewey < prefix(Q.dewey)||0xFF` (see Section 6.2.2). Note that the right side of this condition is an expression. DB2 uses a completely different algorithm for cardinality estimation of expression-based conditions. This algorithm happens to produce a low estimate causing the better plan to be chosen by the optimizer. In fact, when we modified our Global Order queries to include the following condition: `E.end_descendant_id <= Q.end_descendant_id+0`, we observed the same dramatic improvement as with the "tuned" queries.

The message to take away from this section is that, because of costing issues, relational optimizers probably have to be extended to understand XML data before they can be truly effective for querying XML documents.

7.6 Insert Performance

We now compare the insert performance of the three encoding methods under Edge, using both conservative and optimistic insertion techniques (see Section 6.1.6). We used the insertion query where a node is inserted after the second `act` of a single `play` (Hamlet); that is, the XPath selection query was `/play/act[2]`. Two cases are possible. In the first case, the insertion creates no numbering conflicts, and the new element can be immediately inserted. In the second case, the insertion creates a numbering conflict, and requires that some existing nodes be renumbered before the new node can be inserted.

The performance of the optimistic and conservative methods (with and without conflicts) is shown in Table 3. Clearly, the optimistic implementation is much faster in the no-conflict case, since it does not perform conflict detection. Further, the two implementations performed similarly when a conflict was present (and renumbering is required). Thus, the optimistic implementation seems to be a better choice for inserts.

Comparing insertion performance across the order encoding methods, we notice that Local Order resulted in the best overall performance because it has the least renumbering cost (see Section 4.2). Global and Dewey Order show comparable performance in the no-conflict case, but Dewey Order is much faster when there is a conflict. This is because a smaller number of nodes have to be renumbered for Dewey Order as compared to Global Order.

7.7 Reducing the Buffer Size

To evaluate the performance of an RDBMS on a data set that is larger than the buffer size, we reduced the buffer and the sort area to 10M each. We observed that the running times of most selection queries increased by a relatively small amount (<25%) whereas the running times for reconstruction queries increased by a factor of 2 to 5 [17]. Global Order was still the most efficient order encoding method and Dewey Order was the second best. The performance of Local Order degraded more than that of the other two methods because Local Order resorts to sorting very often. We also observed that Inlining is more robust than Edge with a smaller buffer. The performance advantage of Inlining over Edge is more pronounced in that case.

7.8 Comparison with a Main-Memory XPath Processor

We used Microsoft's MSXML to compare the performance of a main-memory XPath processor with the performance of our DB2 implementation. With MSXML, an XML document has to be parsed and converted to a DOM tree before XPath queries can be evaluated. Once the DOM tree has been built, MSXML was up to a factor of 2 faster than DB2 in evaluating most queries [17]. This is because a DOM tree is a ordered, main-memory tree structure. As a result, the ordered axes of XPath can be evaluated without sorting. DB2, however, was 25% faster on Q2 (which includes '//') because it could use the path table to find matching nodes faster.

On the surface, these results may seem negative, but we believe that they should actually be viewed positively. The results show that a general-purpose RDBMS, which can scale to arbitrarily

Table 3: Insert performance.

Order scheme	*Conservative*		*Optimistic*	
	No Conflict	*Conflict*	*No Conflict*	*Conflict*
Global	116.2 ms	2511.9 ms	22.9 ms	2720.2 ms
Local	103.8 ms	176.3 ms	23.0 ms	115.9 ms
Dewey	137.0 ms	1303.6 ms	32.4 ms	1331.1 ms

large document collections, can offer performance in the same ballpark as a specialized, main-memory XPath processor, which can process only a single document at a time. Moreover, the RDBMS does not incur the overhead of parsing a document each time. In our experiments, we found that parsing overhead was often an order of magnitude more expensive than XPath processing.

Summary of Experimental Results

Our experimental results point to the following conclusions:

1. *Relational database systems can support ordered XML queries efficiently.* Our experiments show that relatively complex ordered XML queries can be evaluated over 100MB of data in 2-3 seconds. Even reconstructing the entire XML dataset from the shredded tuples (which has been shown to be expensive even for unordered reconstruction [5][7][12]) only takes on the order of 30-45 seconds.

2. *Global Order is best for query-mostly workloads, while Dewey Order is best for a mix of queries and updates.* The global numbering provided by Global Order makes it efficient for most queries. Dewey Order is slightly less efficient than Global Order due to the higher cost of comparing Dewey paths. Dewey Order has a smaller update overhead, however, which makes it a better option when updates are frequent.

3. *Schema information makes Local Order a viable alternative.* When schema information is available, there is no need to issue (inefficient) recursive queries in order to create a global numbering from Local Order. So the performance of Local Order is comparable (though not at the same level) as Global and Dewey Order. Given Local Order's excellent update characteristics, it may thus be a viable option in update-intensive environments.

4. *The relational optimizers need to understand the hierarchical structure of XML.* The relational optimizer sometimes made wrong estimations, and consequently chose bad plans, when dealing with containment queries. While we were able to "hand tune" the relational optimizer to make it pick reasonable plans, a better (and more general) solution is to extend the optimizer to support the hierarchical XML data.

8. Conclusion

Order is a salient feature of the XML data model, yet it has been largely overlooked in previous on using a relational database system to store and query XML documents. Managing *ordered* XML data using the *unordered* relational model presents new challenges. In this paper, we showed that XML's ordered data model can be efficiently supported by a relational database system. We described three general order encoding methods (*Global Order*, *Local Order*, and *Dewey Order*) for representing XML order in the relational data model, and showed how they can be used to preserve document order during XML query processing. We also described algorithms to translate ordered XML queries into SQL for each order encoding method under two well-known approaches for shredding XML documents into relations (*Edge* [7] and *Inlining* [13]).

The results of an experimental study were presented to evaluate the performance of the three order encoding methods on a real-life data set. Ordered XPath and XQuery queries were used to measure query performance, while XML element insertions were used to measure update performance. Our results show that a relational database system can efficiently support most ordered XML queries. Our experiments also showed that Global Order performs best on query-mostly workloads, while Dewey Order performs best on a mix of queries and updates.

Finally, our experiments showed that, in some cases, current relational optimizers have difficulty in accurately estimating the cost of XML query processing. This leads to very poor plan choices on some XML queries. The problem is that relational optimizers do not understand XML's hierarchical structure. Extending relational optimizers to understand XML hierarchies is an interesting direction for future work.

References

[1] P. Bohannon, J. Freire,P. Roy, J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. *ICDE* 2002.

[2] M. Carey et al., XPERANTO: Publishing Object-Relational Data as XML. In *Workshop on Web and Databases (WebDB)*, 2000.

[3] B. Cooper et al., A Fast Index for Semistructured Data. In *Proc. of VLDB Conference*, 2001.

[4] A. Deutsch, M. Fernandez, D. Suciu. Storing Semistructured Data with STORED. In *Proc. of SIGMOD Conference*, 1999.

[5] M. F. Fernandez, A. Morishima, D. Suciu. Efficient Evaluation of XML Middle-ware Queries. In *SIGMOD*, 2001.

[6] M. F. Fernandez, et al. Publishing Relational Data as XML: The SilkRoute Approach. *IEEE Data Engineering Bulletin* 24(2), 2001.

[7] D. Florescu, D. Kossmann, Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin* 22(3), 1999.

[8] D. D. Kha, M. Yoshikawa, S. Uemura. An XML Indexing Structure with Relative Region Coordinate. In *ICDE* 2001.

[9] Online Computer Library Center. Introduction to the Dewey Decimal Classification. http://www.oclc.org/oclc/fp/about/about_the_ddc.htm.

[10] P. Seshadri, M. Livny, R. Ramakrishnan. Sequence Query Processing. In *Prof. SIGMOD Conference*, 1994.

[11] J. Shanmugasundaram et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. *VLDB* 1999.

[12] J. Shanmugasundaram et al. Efficiently Publishing Relational Data as XML Documents. In *VLDB* 2000.

[13] J. Shanmugasundaram et al. A General Technique for Querying XML Documents using a Relational Database System. *SIGMOD Record*, September 2001.

[14] T. Shimura, M. Yoshikawa, S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Proc. of DEXA Conference*, 1999.

[15] R. Snodgrass, I. Ahn, A Taxonomy of Time in Databases. In *Proc. of SIGMOD Conference*, 1985.

[16] I. Tatarinov, Z. G. Ives, A. Y. Halevy, D. S. Weld. Updating XML. In *Proc. of SIGMOD Conference*, 2001.

[17] I. Tatarinov, et al. Storing and Querying Ordered XML using a Relational DBMS. Tech Report, Univ. of Washington, 2002.

[18] The Plays of Shakespeare in XML. http://www.oasis-open.org/cover/bosakShakespeare200.html.

[19] World Wide Web Consortium. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation Sept. 2001.

[20] World Wide Web Consortium, Extensible Markup Language (XML). W3C Recommendation, February 1998.

[21] World Wide Web Consortium. XML Path Language (XPath), Version 1.0, W3C Recommendation, November 1999.

[22] World Wide Web Consortium. XQuery: A Query Language for XML. W3C Working Draft, June 2001.

[23] F. Yergeau, UTF-8, A Transformation Format of ISO 10646. *Request for Comments 2279*, January 1998.

[24] C. Zhang et al., On Supporting Containment Queries in Relational Database Management Systems. In. *SIGMOD* 2001.

[25] M. Yoshikawa et al., XREL: A Path-Based Approach to Storage and Retrieval of XML documents using Relational Databases. In *ACM Transactions on Internet Technology*, August 2001.