Mixed Mode XML Query Processing

Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy,
Ajith Nagaraja Rao, Feng Tian, Stratis D. Viglas, Yuan Wang, Jeffrey F. Naughton, David J. DeWitt
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
USA
{alanh, bolo, lgalanis, akini, sekar, ajith, ftian, stratis, yuanwang, naughton, dewitt}@cs.wisc.edu

ABSTRACT

Querying XML documents typically involves both tree-based navigation and pattern matching similar to that used in structured information retrieval domains. In this paper, we show that for good performance, a native XML query processing system should support query plans that mix these two processing paradigms. We describe our prototype native XML system, and report on experiments demonstrating that even for simple queries, there are a number of options for how to combine tree-based navigation and structural joins based on information retrieval-style inverted lists, and that these options can have widely varying performance. We present ways of transparently using both techniques in a single system, and provide a cost model for identifying efficient combinations of the techniques. Our preliminary experimental results prove the viability of our approach.

1. INTRODUCTION

As the number of XML documents increases, the importance of building and querying native XML repositories becomes evident. An interesting and challenging aspect of such repositories is that part of the query evaluation process is the discovery of relevant data in addition to its retrieval. This discovery operation often requires a form of simple pattern matching: that is, it requires operations like “find all elements *x* containing a string *s*”, or “find all elements *x* that have an element *y* as an ancestor.” To solve this problem, the database community utilizes inverted list filtering, since the problem is so similar to that addressed in structured information retrieval applications. In addition to inverted list filtering, XML query processing naturally includes navigational access to XML data. Such access is similar to that provided by a DOM interface; here the common operations include finding the children of a given node, or iterating through a set of descendants by doing a depth-first search of the subtree rooted at a given node. In the XML query processing literature to date, there has been a sharp line demarcating the use of inverted list filtering and tree navigation. The purpose of this paper is to show that building systems that keep the two kinds of processing separate is suboptimal, and that by tightly integrating the two types of processing, one can obtain faster query response times. We show this using the Niagara native XML database system.

In more detail, as we will show, there are queries for which inverted list filtering techniques alone are best; there are other queries for which structural navigation techniques alone are best; there are still other queries for which inverted list filtering techniques followed by structural navigation is best; and, perhaps most surprisingly, there are queries for which structural navigation followed by inverted list filtering is best. This suggests that a native XML repository needs to support query plans that utilize these query processing approaches, and needs to be able to pipe intermediate results between the two. Finally, given that no one style of processing dominates, an XML query processor requires query optimization techniques and statistics to decide how to choose among the alternatives for any given query.

This paper makes the following contributions:

- We present the main structure of a scalable system for storing and querying static XML data. In particular, we explain in detail the approach used in two key parts, a structure index module called the Index Manager and a data storage module called the Data Manager. The Niagara overview [19] describes the system architecture in general terms, but the presentation in that paper did not provide sufficient detail to motivate the tight coupling of the Index Manager and the Data Manager.
- We present algorithms for answering queries using either module, along with a cost model for each

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

algorithm. The cost model is dependent on statistics capturing the structure of XML documents, and we propose new statistics necessary to ensure cost model accuracy.

- We present a decision algorithm using the proposed cost model to decide which combination of query processing techniques (inverted list filtering and/or tree navigation) should be used for a given query over a specific dataset.
- We present an experimental study of this framework.

The rest of the paper is organized as follows: An overview of the system and the relevant modules is presented in Section 2. The specific algorithms for using each of the modules to process queries over XML data, the costs associated with each algorithm, and a decision process selecting the correct algorithm are presented in Section 3. An experimental evaluation of the proposed approach is presented in Section 4. A discussion of related work is contained in Section 5. Finally, the conclusions and the future work directions are summarized in Section 6.

2. SYSTEM OVERVIEW

2.1 System Architecture

Our system is perhaps best described by examining how it processes and stores an XML document. As shown in Figure 1, the process of loading an XML document begins by running it through a front-end parser. The parsed XML is then fed into the Data Manager and the Index Manager.

- The *Data Manager* stores a DOM-style tree representation of the XML document.
- The *Index Manager* stores a set of inverted lists, mapping elements, attributes, and words found in the XML document to lists of exact locations within the document.

The Shore Storage Manager [6] is used for storage and indexing.

Once a set of documents has been loaded into the Data Manager and the Index Manager, the system is ready to execute queries over those documents. To support this, our system provides a query parser for XQuery, an optimizer, and a tuple-based execution engine.

The system is designed with scalability and performance in mind. To that end, physical operators within our query execution engine are executed on separate threads, and communicate with each other via a message queue. This allows parallel execution of operators as well as a straightforward extension to a distributed execution environment.

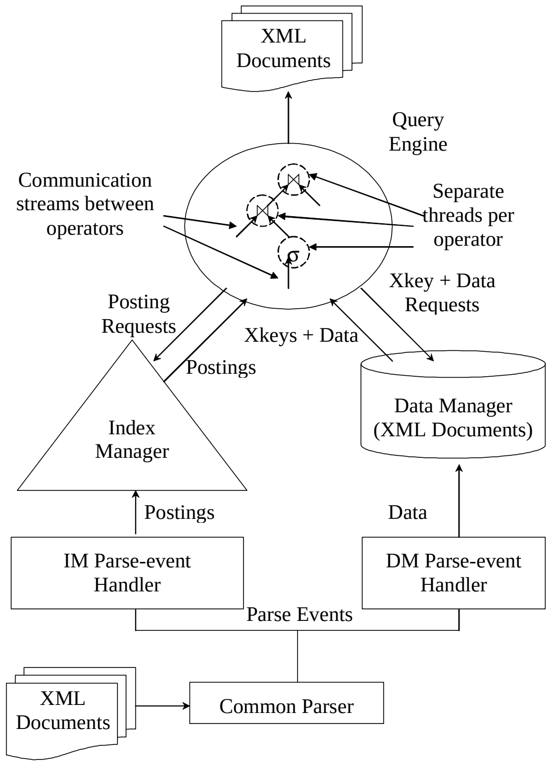

Figure 1 – Basic System Architecture

2.1.1 Numbering Scheme

To facilitate mixed-mode query processing, the Data Manager and Index Manager must share a common scheme for numbering the elements in an XML document. For performing structural joins using inverted lists, the results in [2,16,25] have demonstrated that assigning a start number, end number, and level to each element suffices. Each element in the Data Manager is uniquely identified by its start number and the id of the document containing the element. An example XML document showing the start and end number assignments for each element appears in Figure 2. Additional numbers are assigned to attributes and words occurring in attribute values and element contents. We omit these details as they are not relevant to the focus of this paper.

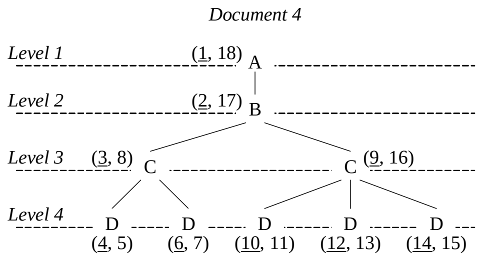

Figure 2 - Numbered XML Document

2.1.2 Data Manager

Each XML document is stored in the Data Manager using a B+-tree structure. Figure 3 illustrates this structure for the example document in Figure 2. The key of the B+-tree index is a (document id, element id) pair that we refer to as an XKey. In addition to an XKey, each leaf entry in the B+-tree contains:

- **Term id** – The element name converted to an id number via a hash table.
- **A Record id** (RID).

This RID specifies the address of a record in the data manager which contains the following information about the element:

- **End number, Level**
- **Element Payload** – The actual text for the element
- **A list of (term id, element id) pairs** – All children of the element, in document order

Attributes are stored in the leaves of the B+-tree following their enclosing element.

The leaf level of the B+-tree shown in Figure 3 has nine entries, corresponding to the nine elements of the XML document from Figure 2. Consider the leaf entry corresponding to the B element. It is comprised of the XKey and the term id, which is ((4,2),26), and the rid. The corresponding record has the end number and level (17,2), and a list of child elements.

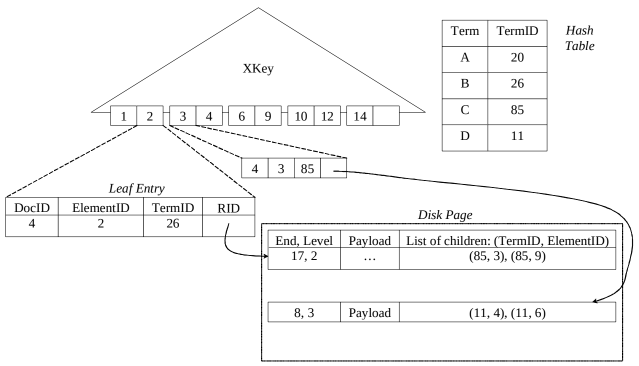

Figure 3 - Data Manager tree structure

The Data Manager provides a DOM-like interface to its clients. Two types of cursors are supported:

- The ***Child Axis (CA)* cursor** takes as input an XKey and an optional tag name, and enumerates the children of that element in document order. If a tag name is specified, only child elements with a matching tag name are returned by the cursor.
- The ***Descendent Axis (DA)* cursor** also takes an XKey and an optional tag name as input, but enumerates all proper descendant elements. Element name filtering based on the input tag name occurs in this case also.

2.1.3 Index Manager

In order to be able to efficiently identify XML documents that contain a particular term [15,19], the Index Manager maintains posting lists describing the occurrence of elements, attributes, and text for the documents stored in the Data Manager. We next describe how this information is structured to facilitate the scalable retrieval of term locations both on an intra- and inter-document basis.

This indexing information is stored in a two level index structure. The top level index is a B+-tree with (term id, doc id) as the key. The value associated with each leaf entry is an inverted list that contains information about occurrences of a particular term in a particular document. Each occurrence is represented by a start number, end number, and level triple as proposed in [2,16,25]. We refer to this info as a *posting*, and the entire list as a *posting list*.

The second level index is built on each posting list. This index consists of a single index page stored in the leaf level of the top level B+-tree. For each page in the posting list, the index page has an entry with the start number of the first posting in that page. When the cardinality of the posting list is very small, we inline the posting list in the top level B+-tree leaf level pages instead of using a second level index. Similarly, when the number of postings becomes so large that the second level index no longer fits on a single page, we switch to a separate B+-tree index for this posting list. **Figure 4** illustrates how the document in Figure 2 would be indexed.

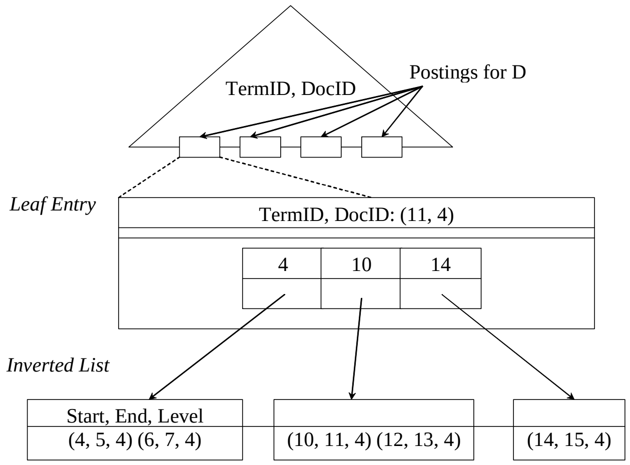

Figure 4 - Index Manager tree structure

To find all occurrences of a term in a repository of documents, the system performs a range scan on the top level B+-tree using the term id as a partial key. To find all occurrences in a single document, the pair (term id, doc id) is used as the search key to retrieve the entire posting list. As will be demonstrated later, being able to efficiently offset into the posting list for a particular document using a start number will be beneficial for our structural join algorithm. We support this through the use of the second level index.

3. MIXED MODE QUERY PROCESSING

In this section, we first describe the relevant path expression evaluation algorithms of our system. We develop a cost model which estimates the cost for an execution strategy for each algorithm given a set of statistics. Finally, we conclude this section by describing our plan enumeration strategy.

3.1 Notation

Our cost model depends on several statistics and cost estimates of fundamental operations. The path expression statistics are maintained on both a per document basis and across all documents. Table 1 provides a list of common notations used throughout this paper and explanations for each. Note that the path expression statistics used can be computed with any of the XML summary structures that have been proposed in the literature [1,7,12,20,23], with the exception of the skip factor SF(A,B) and the skip count SC(A,B). A possible strategy for gathering these statistics is discussed in section 3.3.2.

Table 1 – Notation used in cost formulas

|A|	Cardinality of element A
|A/B|, |A//B|	Number of B elements that have an A parent (/) or A ancestor (//) – B can be '*' to count all children/descendants of A
cac	Time to open a child axis cursor in the Data Manager (including I/O)
dac	Time to open a descendant axis cursor in the Data Manager (including I/O)
EBP	Number of element entries per leaf page in the Data Manager B+-tree
PBP	Number of postings per Index Manager backing store page
F	B+-tree lookup cost (including I/O)
IO	Cost for a single page I/O
OC	Communication cost between operators per XKey or posting
comp	Time to compare integers in main memory
{P1 | P2}	Average number of P1 paths for an element satisfying the context path P2
$\{P1 \mid P2\}_{NL}$	Average number of P1 paths, which terminate on a non-leaf element, for an element satisfying the context path P2
SF(A,B)	The fraction of comparisons which can be skipped when processing A/B or A//B
SC(A,B)	The count of skips that occur when processing A/B or A//B

3.2 Data Manager

The Data Manager supports navigation-based algorithms for evaluating path expression queries. This section presents one such algorithm, which we call Unnest.

3.2.1 Unnest Algorithm

The Unnest algorithm takes as input a path expression and a stream of XKeys. It evaluates the path expression for each XKey in the input, and outputs XKeys corresponding to the satisfying elements.

As an example, consider the path expression document("*")/A/B/C. This path expression should return all C elements matching the /A/B/C path from all documents loaded and indexed by the system. To evaluate this path expression, we create an Unnest operator with A/B/C as the associated path expression query. A list of XKeys for the root elements of all documents stored in the Data Manager is the input for this operator. The algorithm then applies the path expression to each of the root elements, and returns the satisfying C element XKeys.

We next describe the general algorithm that Unnest uses to process path expressions using two specific examples. The Unnest algorithm uses a Finite State Machine (FSM) to evaluate path expressions. Each state of the FSM represents having satisfied some prefix of the path expression, while an accepting state indicates a full match. Each state is also associated with a cursor that corresponds to the next step to be applied for the path expression. For each XKey obtained from the cursor, we make the appropriate transition in the FSM. We then continue with the new XKey in the new state. Upon termination of a cursor, we return to the previous state and continue enumerating its cursor.

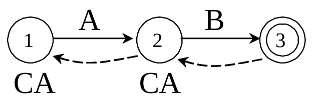

Figure 5 – Unnest FSM for A/B

Consider the path expression A/B. The corresponding FSM is given in Figure 5. This figure shows a simple FSM which accepts paths of the form A/B – that is, B elements which have an A parent. State 1 is the start state. For each input XKey, a ***CA*** cursor is opened on term name A. For each element returned by this cursor, we transition to state 2. In state 2, we open another ***CA*** cursor with term name B. For each B element in this cursor, we transition to state 3, which is an accepting state. We then output the B element XKey and return to state 2 to finish the ***CA*** cursor enumeration. Similarly, we must return to state 1 whenever a state 2 ***CA*** cursor enumeration terminates, and continue the ***CA*** cursor enumeration there.

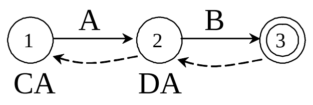

Figure 6 - Unnest DA-FSM for A//B

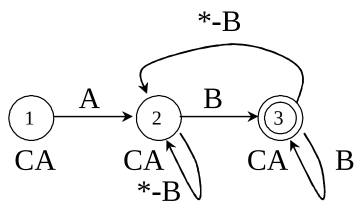

Figure 7 - Unnest CA-FSM for A//B

Therefore, evaluating a ***CA*** path is quite straightforward. In order to handle a descendant axis path expression such as A//B, however, we choose among two possible state machines. For this path, Figure 6 shows a deterministic FSM that utilizes a ***DA*** cursor (***DA***-FSM), and Figure 7 shows a non-deterministic FSM that uses only ***CA*** cursors (***CA***-FSM). We convert the non-deterministic FSM to a deterministic FSM before evaluation using a standard NFA to DFA conversion algorithm. The resulting DFA also uses ***CA*** cursors only.

Each of the two solutions for evaluating a ***DA*** path using Unnest has its own advantages and disadvantages. The ***DA***-FSM is a straightforward representation of the A//B query. Most of the work in this case is pushed down to the data manager through the use of a ***DA*** cursor. On the other hand, the ***CA***-FSM opens a ***CA*** cursor for every non-leaf descendant element of each satisfying A element. Notice how in the former case, a single scan of the leaf of the B+-tree by the ***DA*** cursor identifies all satisfying B elements, while in the latter case a much larger per descendant overhead is incurred.

In certain cases, evaluating the ***DA***-FSM on path expressions may perform unnecessary computation and produce duplicate results. For example, consider the query Q=A//B//C. Recall that the result of this query according to XQuery semantics is the set of C elements satisfying Q. Suppose the XML document contains the path A/B/B/C/C. Although each C has more than one A//B ancestor path, it should appear only once in the result. Using ***DA*** cursors, each C element will be output twice, once for each B ancestor. A *distinct* operator is required to remove duplicates from this result. Moreover, the subtree under the second B will be examined twice during query evaluation. By using the ***CA***-FSM for this query, duplicate-free results can be produced while avoiding unnecessary reexamination of parts of the data. In this case, the comparison between ***CA***-FSM and ***DA***-FSM is similar to the comparison between stack-based and merge-based algorithms for evaluating structural joins [2].

Even for simple queries like A//B/C for which the ***DA***-FSM is guaranteed to produce duplicate-free results, the list of results may not be in document order. For example, this can happen for the query above when a B element parents another B element, and each B parents a C element as its last child. The ***CA***-FSM in contrast will always produce results in document order. This may be a factor if document order results are required either by the query or an upper-level operator like a structural join.

3.2.2 Cost Model

We present two relevant cost formulas in this section – the cost of a child axis unnest, and the cost of a descendant axis unnest.

Let us now consider the costs of using cursors in more detail. Opening a Child Axis cursor involves navigating the B+-tree and following the rid to get the children list for the element. Enumerating all satisfying elements

$\text{Cost}_{\text{Unnest}}() = OC$	This represents the cost of outputting a single XKey from an Unnest operator, and gives us a base case for stopping recursion.
$\text{Cost}_{\text{Unnest}}(./A/P1) = cac$	For an input element, we must open a child access cursor
$+ \{/* \mid .\} * 2 * comp$	Examine each of the average number of children for a single input element
$+ \{/A \mid .\} * \text{Cost}_{\text{Unnest}}(P1)$	For each A child of the input, we must pay the cost of unnesting the rest of the path
$\text{Cost}_{\text{Unnest}}(.//A/P1) = \min($	Choose the best ***DA*** plan
$(dac$	For an input element, we must open a descendant access cursor
$+ \{//* \mid .\} * 2 * comp$	Examine each of the average number of descendants for a single
$+ \lceil \{//* \mid .\} / EBP \rceil * IO$	input element, factoring in the I/O cost for the leaf pages loaded
$+ \{//A \mid .\} * \text{Cost}_{\text{Unnest}}(P1)),$	For each A descendant of an input, we must pay the cost of unnesting the rest of the path
$(cac$	For an input element, we must open a child access cursor
$+ \{//* \mid .\} * 2 * comp$	Examine each of the average number of descendants for a single input element
$+ \{//* \mid .\}_{NL} * cac$	For each non-leaf descendant of a single input element, we must open a child access cursor
$+ \lvert .//A/P1 \rvert * \text{Cost}_{\text{Unnest}}())$	Cost of outputting the result
$)$	

Equation 1: The cost of Unnest

involves a traversal of the children list.

To open a Descendant Axis cursor, we follow the same path to find the element information. Enumerating all satisfying elements involves a leaf-level scan of the B+-tree of all descendants of this element.

Each cost formula is defined recursively. The cost of this algorithm is given in Equation 1. We omit the potential costs of duplicate elimination and document order sorting from the cost formulae for readability.

3.3 Index Manager

3.3.1 ZigZag Join Algorithm

A/B and A//B paths are processed in the Index Manager using the ZigZag Join algorithm. This algorithm is a natural extension of the Multi-Predicate Merge Join (MPMGJN) algorithm described in [25] to make use of the indices present on the posting lists. A similar algorithm was recently proposed in [8]. These algorithms assume that the A and B posting lists are sorted in order by (document id, start number).

The MPMGJN algorithm optimizes the backtracking step by never reconsidering the postings in one list that are guaranteed not to have any further matches in the other list. The main extension our algorithm provides is to use the index on the postings to skip forward over parts of a posting list that are guaranteed not to have any matching postings on the other list. For example, consider the evaluation of the query A//B over the XML document represented by Figure 8. The ZigZag Join algorithm checks the containment of the first B within the first A, and outputs the pair. It then advances the B posting list pointer, and finds that the second B is not contained by the first A, causing an advance of the A posting list pointer. When the algorithm discovers that the second A is beyond the second B, it needs to advance the B posting list pointer. Since the current B posting had no matching A posting, it uses the second level index to seek forward using the current A posting's start number. In this case, it skips over two B postings to the fifth B posting. For this example, we were able to use the index to skip parts of the descendant (B) posting list. In a similar fashion, we may be able to skip parts of the ancestor posting list as well.

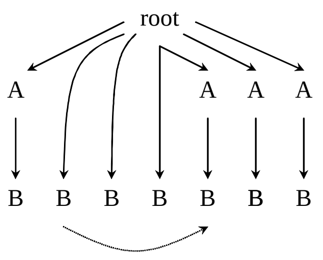

Figure 8 - ZigZag of A//B will attempt 1 skip

The above example involved a single document. In general, the skipping is done across documents by using the (term id, doc id) pair. The index can be used when one or both posting lists are scanned directly from the Index Manager. For the case when an input posting list has been created by a previous operator, we maintain a dynamic one level index on this posting list and utilize this index to perform the skipping. We must also buffer a posting in the posting list until the algorithm identifies that it will no longer backtrack to this posting.

3.3.2 Cost Model

Determining an accurate cost model for the ZigZag Join is somewhat complicated. Because the algorithm can "skip" over sections of either input posting list and can backtrack in a complex fashion, the CPU cost can be quite dependent on actual document structure. In the best case we may only need to do O(|A//B|) comparisons of start and end numbers. In the worst case, we may have to perform O(|A|*|B|) comparisons.

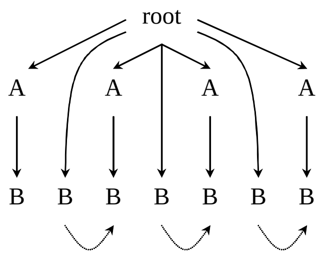

Figure 9 - ZigZag of A//B will attempt 3 skips

Two factors need to be considered for properly estimating the cost of the ZigZag Join algorithm. The first is the percentage of comparisons avoided by efficient backtracking and forward skipping using the second level index. The second is a total count of the index lookups to seek forward. We next give an example to show why the latter is required. Consider the two XML documents represented by Figure 8 and Figure 9. Both documents have exactly the same number of A elements, B elements, and A//B paths. However, the distribution of these elements within the document is different. This leads to the algorithm using the B+-tree (to skip forward) once for the document in Figure 8 and three times in Figure 9.

We define the skip factor, SF(A,B), to be the ratio of comparisons avoided by our algorithm to the maximum number of comparisons, that is (|A|*|B|). The skip count, SC(A,B), is defined as the number of second level index lookups performed for purposes of skipping forward by our algorithm. We believe that accurate and efficient computation of these statistics is an interesting and important area for future work. As a simple initial approach, a brute force execution of the ZigZag algorithm for each possible pair of elements in the document will work. We only need to count the number of skips along the way, and directly compute the skip factor at the end. As a trivial optimization, we can avoid running the

$\mathbf{Cost_{ZigZag} = 2 * F * comp}$	Cost of index lookup for 1st A and B.
$\mathbf{+ (\|A\| * \|B\| * comp * 2 + (\lceil \|A\| / PBP \rceil + \lceil \|B\| / PBP \rceil) * IO) * (1 - SF(A,B))}$	Cost of comparisons necessary to determine A/B or A//B relationship. This is scaled by the skipping factor to account for the unnecessary comparisons, and includes the I/O cost for loading pages of postings.
$\mathbf{+ \{A//B\|.\} * OC}$	Factor in the cost of outputting all matching B element postings.
$\mathbf{+ SC(A,B) * F * comp * (\{A//B\|.\} / \|A//B\|)}$	When a skip occurs, we go back to the B+-tree to find the next position in the posting list.

Equation 2: The cost of ZigZag join

algorithm for any pair of elements A and B for which |A//B| is zero.

The cost formula for A//B is given by Equation 2. In a similar fashion, we define the cost formula for A/B and other variants where only one of the two postings is projected. Scaling the I/O cost by the Skip Factor is a first level approximation of the potential for avoiding entire page I/Os.

3.4 Enabling Mixed Mode Execution

Recall that the ZigZag Join operator takes posting lists as input, and the Unnest operator takes a list of XKeys as input. To enable query plans that use a mixture of these two operators, we must provide efficient mechanisms for switching between the two formats.

Converting a list of postings into XKeys is as simple as removing the end number and level. This is possible since the start number and element id for a given element are identical in our numbering scheme.

On the other hand, in order to convert an XKey into a posting, we need to look up an end number and level. To support this operation, we store the end number and level in the information record for each element. A simple B+-tree lookup followed by a potential I/O to retrieve this page is therefore required to perform the conversion from an XKey to a posting. As an alternate approach, we could include the end number with the entries in the child list of each element. The conversion of XKeys to postings would benefit from this at the expense of increasing the Data Manager storage requirements. We explore this issue more fully in Section 4.3.

3.5 Selecting a Plan

Given a path expression query, let us now look at how we can combine the ZigZag Join and Unnest algorithms to produce alternate query plans. Recall that the ZigZag Join algorithm executes one step of the path expression query. The Unnest algorithm can execute one or more steps using a single FSM.

We heuristically limit our search space to include only left-deep evaluation plans for structural joins. To choose the best plan, we use a dynamic programming approach. For a path expression query, the cost can be expressed as the sum of cost of the last operation and the minimum cost for the rest of the path expression. For example, consider the query /A/B/D//F.

- If the last operation is a ZigZag Join, then it corresponds to the operation D//F. So, the cost of the query is the ZigZag cost of D//F plus the minimum cost for evaluating /A/B/D.
- If the last operation is an Unnest, then it may correspond to one of the proper suffixes of the path expression. We must consider the cost of Unnest for .//F, ./D//F, ./B/D//F, and /A/B/D//F, adding to each the minimum cost of evaluating the corresponding prefix.

Table 2 – Sample Cost Calculation Matrix for Unnest

I_U	**0**	**1** */A*	**2** */B*	**3** */D*	**4** *//F*
0	0	24025	48047	72869	9910886
1 /A	X	X	48050	72872	9910889
2 /B	X	X	X	72871	9910888
3 /D	X	X	X	X	9910889
4 //F	X	X	X	X	X

Table 3 – Sample Cost Calculation Matrix For ZigZag Join

I_Z	**0**	**1** */A*	**2** */B*	**3** */D*	**4** *//F*
0	0	38042	48042	X	X
1 /A	X	X	86084	X	X
2 /B	X	X	X	4290885	X
3 /D	X	X	X	X	146932
4 //F	X	X	X	X	X

Given a path expression with N elements, we construct two (N+1) x (N+1) matrices – one each for Unnest (I_U) and ZigZag Join (I_Z). We maintain the costs for each algorithm separately to account for the possible penalties incurred due to changing formats in mixed mode execution. We will explain the process on the example query /A/B/D//F. The corresponding matrices are shown in Table 2 and Table 3. We create a 5x5 matrix in this case. For each cell in the matrix, we calculate the minimum cost for evaluating the prefix of the path expression along the X axis, given a prefix along the Y axis as the input. For example, the gray square in the I_Z matrix ($I_Z(4,3)$) is the minimum cost for having used ZigZag Join to evaluate .//F given that /A/B/D is our input. Similarly, $I_U(3,0)$ is the minimum cost for

subpath(P,m,n)	Given path P, extract a partial path starting with the m'th element, extending n elements and including the leading path axis
$\mathbf{I_U(x, y) = \lvert subpath(P, 1, y)\rvert * Cost_{Unnest}(subpath(P, y+1, x-y+1))}$	The cost of running Unnest over the next subpath for all input XKeys which were output from a length y prefix
$\mathbf{+ \min (\min_{0<=j<y}(I_U(y, j)) , I_Z(y,y-1) + \lvert subpath(P, 1, y)\rvert * (comp+OC))}$	The best subplan which evaluates the length y prefix, taking into account the cost of converting postings to XKeys
$\mathbf{I_Z(x,y) = Cost_{ZigZag}(subpath(P, y,1), subpath(P, x, 1))}$	The cost of running ZigZag Join given a list of postings which were output from a length y prefix
$\mathbf{+ \min(\min_{0<=z<y}(I_U(y, z)) + \lvert subpath(P, 1, y)\rvert * (F+OC) , I_Z(y,y-1))}$	The best subplan which evaluates the length y prefix, factoring in the XKey->posting conversion cost
$\mathbf{\lvert subpath(P,1,0)\rvert = 1}$	Initialization steps
$\mathbf{I_Z(0,0) = I_U(0,0) = 0}$	

Equation 3: Choosing a plan

evaluating /A/B/D in a single Unnest operator with the root of the document as the single input element.

We use an 'X' to show cells within each matrix that do not need to be calculated. For example, the diagonal of each matrix and values below the diagonal are not of interest. Cell $I_Z(2,0)$ refers to a single ZigZag Join operator with A and B posting scans as the left and right inputs, respectively. Here, a check on the level number for A postings is performed in the ZigZag Join to ensure that only root A elements are chosen. Cell $I_Z(2,1)$ still has a B posting scan as the right input, but the left input is the output of any operator capable of having evaluated /A.

To calculate the value for each interesting cell in Tables 2 and 3, we define the formulae of Equation 3. Notice that the cost of the optimal plan is the minimum cost in the 4th column (corresponding to //F) of I_U and I_Z.

4. EXPERIMENTS

This section presents experimental results to validate the necessity and viability of the mixed mode query processing approach.

All experiments were executed on a dual processor 550 MHz Pentium III PC running RedHat Linux 6.2, equipped with 1GB of main memory with SCSI disks. A single 8GB disk for storing both the Shore log and database volume was utilized. All queries are read only, so no logging occurs during query execution. The buffer pool size was set to 64MB throughout our experiments. All timings reported in this section are an average of 10 runs. We calculated that all timings for each average are within 1% of the average value with 99% confidence.

The experiments conducted used the XML Schema of Figure 10. Three documents of varying sizes were generated in the following manner. The schema of Figure 10 contains four *-edges. By fixing the average fan-out of each *-edge, the width of a document conforming to the schema can be varied. For the smallest document, the average fan-outs of the B/C, C/D, D/E and E/G edges were set to 4, 256, 4, and 256, respectively. By increasing the average fan-outs of the B/C and D/E edges to 40 and then to 400, two new documents were obtained. To reflect the relative sizes of the three documents, the expected number of D elements in each is used as an identifier. The statistics about the documents are presented in Table 4. For all documents the average number of keywords per PCDATA element was set to 8.

Table 4 - Synthetic document statistics

Document	*Actual number of D elements*	*Total number of elements*	*File size (KB)*
1K	1143	2140	116
10K	11676	21935	1,182
100K	107807	216666	11,588

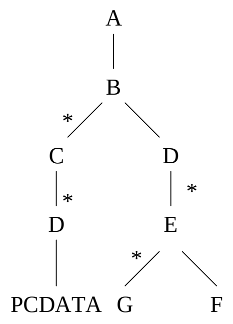

Figure 10 - Test XML Document Schema

Table 5 shows the four queries that were used along with the expected optimal evaluation plan for each query. The four queries were selected to illustrate a scenario where a particular evaluation strategy dominates. We now explain our notation for representing mixed mode plans in this section. Consider the predicted optimal plan for Query 3. This corresponds to a single Unnest operator evaluating

/A/B/D, and feeding a ZigZag Join evaluating D//F. PostingScan(//D) refers to a simple scan of the entire posting list for element D.

Table 5 - Test Queries with predicted optimal plans

Number	*Query*	*Predicted optimal plan*
1	/A/B/D	Unnest(A/B/D)
2	//B/D	PostingScan(//B) + Unnest(./D)
3	/A/B/D//F	Unnest(/A/B/D) + ZigZag(//F)
4	//D	PostingScan(//D)

In the case of a query over a single document, if the bottommost operator is a ZigZag Join, the doc id is passed to the join to restrict the computation to the required document. If the bottommost operator is an Unnest operator instead, the root element of the document is passed as input. On the other hand, for queries over all documents (in-*), if the bottommost operator is Unnest, a list of document root elements is retrieved from the catalog and used as input. No additional work is required when the bottommost operator is a ZigZag Join.

4.1 Mixed Mode Evaluation Experiments

Previous work has argued that structural joins are preferable to navigational style processing for path expression evaluation. In the experiments conducted we present three cases in which the optimal evaluation took place either entirely or in part in the Data Manager. All results in this section refer to single document queries. Please refer to section 4.3 for scalability results.

The execution times of four alternate plans for Query 1 are given in Table 6 (all times in Tables 6 through 9 are with a cold buffer pool). In this section, we refer to an Unnest operator as UN and a ZigZag Join operator as ZZ. The UN(/A/B/D) plan offers the best performance among the four plans considered. The gap between this plan and the others widens considerably as the size of the document is increased. The intuition behind this result is as follows. The Unnest operator only considers the A element, the B element, and all the children of B. Even for the document containing 100K D elements, the total number of elements considered by Unnest is under 500. On the other hand, a ZigZag Join evaluating B/D must consider all of the D descendants of B. This is because any of these D descendants may actually be a child element of a B element. As a result, this algorithm has to consider roughly 100K postings. For the other queries, we only present the two extreme plans and the optimal plan.

In Table 7, we see the execution times for //B/D. The optimal plan is quite fast, regardless of the size of the document. The slight increase is execution time as the document size increases is due to the increasing number of C child elements of B. This query clearly demonstrates the benefits of a mixed mode approach. The leading //B is an expensive operation if performed using the Unnest operator, but comparatively cheaper if a posting scan is used instead. The B/D operation, as we saw above, is very cheap if executed by Unnest as compared to the ZigZag Join. Combining these two operators into a hybrid plan offers the optimal performance for executing this query.

Query 3 is very similar to Query 1, but adds a descendant lookup for F for each matching /A/B/D. The corresponding execution times are presented in Table 8. The best plan in this case is predicted to be an Unnest of /A/B/D feeding the left input of a ZigZag Join with an F posting scan on the right. In contrast with Query 2, this query has a descendant axis after the B/D step. As a result, an Unnest followed by a ZigZag Join has the best performance.

Query 4 is a very simple query, and there are only two choices to evaluate it. We can either Unnest //D or run a posting scan for D elements. This query is, of course, the exact query that inverted lists are designed to handle with optimal efficiency. On the other hand, the Unnest operator must examine the entire document to evaluate this query. As illustrated by the results in Table 9 the posting scan provides the best results.

The results in this section show that for varying document sizes, hybrid strategies are worth considering.

Table 6 - Execution times in milliseconds for Query 1

Document Size	*ZZ*	*ZZ→UN*	*UN→ZZ*	*UN*
1K	15.2	9.3	18.1	6.3
10K	102.8	13.6	108.3	10.3
100K	719.0	17.2	728.5	17.6

Table 7 - Execution times in milliseconds for Query 2

Document Size	*ZZ*	*UN*	*ZZ→UN*
1K	10.1	190.8	8.6
10K	78.0	1878.6	13.7
100K	425.3	18471.1	17.2

Table 8 - Execution times in milliseconds for Query 3

Document Size	*ZZ*	*UN*	*UN→ZZ*
1K	18.9	93.7	11.7
10K	110.3	896.8	27.8
100K	749.9	9308.3	57.7

Table 9 - Execution times in milliseconds for Query 4

Document Size	*ZZ*	*UN*
1K	35.4	244.9
10K	323.6	2442.5
100K	2834.3	23641.2

4.2 Cost Model Validation

In this section we compare the predictions of the cost model to the measured performance of each query. The cost estimations were made using the values in Table 10. The values for PBP and EBP are the actual parameters we set for the experiments in Shore. We set IO to be a factor of 10K more expensive than a comparison.

Table 10 - Values for various cost model parameters

IO	*comp*	*EBP*	*PBP*
10000	1	82	256
OC	***F***	***cac***	***dac***
3	20*comp+1.4*IO	F+IO	F+IO

We present our results for Query 1 across various document sizes, and for all queries over the 100K document. The comparisons for cold buffers for Query 1 are shown in Table 11. The entries are normalized to the minimum entry in the corresponding row. Even though the cost model ratios can be off significantly from the actual ratios, the estimated ratios are close enough that an optimizer using our cost estimates orders the plans correctly. For each document size, our cost model orders the four plans correctly. The relative ratios are predicted reasonably and the accuracy increases as the document size increases.

Table 11 – Comparison of Cost Model estimates to Actual costs for Query 1, normalized to the UN plan

		UN	*UN→ZZ*	*ZZ→UN*	*ZZ*
1K	Estimated	1.0	2.0	1.0	1.8
	Actual	1.0	3.0	2.2	2.5
10K	Estimated	1.0	7.2	1.0	7.0
	Actual	1.0	11.0	1.3	10.7
100K	Estimated	1.0	72.8	1.0	58.9
	Actual	1.0	83.5	1.1	83.5

In Table 12 we show a comparison of cost model estimates to actual costs for the four queries evaluated over the 100K document. Even in this case, for each query the cost model arranges the plans in the same order as the actual execution times.

Table 12 - Comparison of Cost Model estimates to Actual costs for the 100K Document, normalized to the best plan for each query

		UN	*UN→ZZ*	*ZZ→UN*	*ZZ*
Q1	Estimated	1.0	59.1	1.0	58.9
	Actual	1.0	173.2	1.3	172.2
Q2	Estimated	566.3		1.0	121.8
	Actual	5134.0		1.0	78.0
Q3	Estimated	67.5	1.0		29.6
	Actual	466.2	1.0		31.6
Q4	Estimated	4.7			1.0
	Actual	8.3			1.0

4.3 Scalability Experiments

All results reported in sections 4.1 and 4.2 are measurements of queries executed over a single document. We now consider the effects of loading several document size distributions and executing the same queries over the entire set of loaded documents. Our results indicate that the mixed mode query processing approach does continue to show benefit as the total number of documents increases. This section details our scalability experiments and results.

For these experiments, we chose to create a set of documents clustered around each of our previous document sizes of 1K, 10K, and 100K. For each of these sizes, we created documents with approximately 50%, 75%, 100%, 125%, and 150% of the original sizes. These documents were then loaded based upon six distributions, as detailed in Table 13. The distributions were chosen in an attempt to keep the total element count of each document set constant. Distribution I consists of the 1K document set only. Using Table 4, we can calculate that Distribution I represents approximately 250 MB of XML data. Similarly, distributions II and III consist of the 10K and 100K document sets, respectively. Distribution IV is a 60%, 30%, 10% D element split, while V gets equal numbers of D elements from each document set. Finally, distribution VI is simply an equal number of each document, and therefore the majority of D elements come from the 100K document set.

Table 13 - Document distributions for the scalability experiments

Distribution	*1K*	*10K*	*100K*
I	2500	0	0
II	0	250	0
III	0	0	25
IV	1500	75	3
V	888	88	8
VI	25	25	25

Our expectation is that the query performance of a distribution comprised of mostly small documents should be similar to that of a query executed over a single document scaled by the number of documents in the distribution. Similarly, we expect queries over distributions with more large documents to have performance more like that of queries over a single large document, again scaled by the total document count. Below in Table 14, we show the results for executing the four query plans for Query 1 over the distributions listed above. As expected, the Unnest plan is the best plan for Query 1, and the ZigZag followed by Unnest is tied for the top spot. It also appears that using the ZigZag join to process the D element list becomes a very poor choice for distributions which contain a large amount of the 100K document cluster. For example, distribution I (2500 1K docs) shows only a 1.5x performance penalty for using ZigZag, while distribution III (25 100K docs) shows a 68.7x penalty. Comparing the results reported here with those in Table 6, we find that a similar effect exists in the single document case as well. A similar pattern can be found in the execution times for Queries 2 and 4, so those tables are omitted for brevity.

One interesting anomaly exists for Query 3, however. For the 2500 1K docs, the ZigZag plan turns in the best execution time, although the Unnest feeding ZigZag plan should win easily. We determined that converting each XKey to a posting required more I/O than expected for this case. With a larger number of documents loaded into the Data Manager, our cost model underestimates the conversion cost. To verify this theory, we modified the format of the element information record stored in the Data Manager to include the end number for each child of the element, and reran the experiments using this new format. The cost of converting XKeys to postings with this new format is very small, and the UN->ZZ plan becomes the best plan as originally predicted. Although this modification allows queries such as query 3 to convert XKeys to postings much more efficiently, it does so at the cost of redundantly storing the end number. This could have the effect of slowing down some Data Manager-only queries due to the possibility of retrieving additional record storage pages during execution.

Table 14 – Cost of executing the four plans for Query 1 over each distribution, normalized to the UN plan

Distribution	*ZZ*	*ZZ->UN*	*UN->ZZ*	*UN*
I	1.5	1.0	4.1	1.0
II	7.2	1.0	25.3	1.0
III	68.7	1.0	199.1	1.0
IV	2.6	1.0	6.7	1.0
V	3.0	1.0	8.6	1.0
VI	27.6	1.0	76.2	1.0

5. RELATED WORK

There has been a lot of work on developing efficient algorithms for structural joins that identify occurrences of structural relationships like ancestor-descendant and parent-child relationships. Using "pre-order" and "post-order" numbers to determine such structural relationships was presented in [10]. In [25], the authors proposed the multi-predicate merge join algorithm (MPMGJN) to efficiently merge two sorted lists. A merge based join algorithm was proposed in [16]. In [2], two families of structural join algorithms were proposed: tree-merge and stack-merge. The tree-merge algorithms were extensions of the traditional merge algorithms. They also showed that the stack-tree algorithms have better worst-case linear guarantees (linear in size of inputs and output) than the tree-merge algorithms. In [8], the authors enhanced the stack-merge algorithms to make use of B-tree indices on the inverted lists. The structural join in this case uses the index to skip those parts of the inverted lists that do not participate in the join. The ZigZag join algorithm presented in this paper is a similar extension to the MPMGJN join in the presence of indices on the inverted lists. One of the stack-merge algorithms (Stack-Tree-Ancestor) is only a partially non-blocking algorithm, while the tree-merge algorithms are all non-blocking. This was one of the reasons why we started with the MPMGJN algorithm. We would like to emphasize the fact that the hybrid strategy presented in this paper works irrespective of the actual structural join algorithm used. Also, since the dataset we used in our experiments does not have any structural recursion in it, the stack-based and merge-based algorithms perform similar number of comparisons. So, even when we use a stack-based structural join, the experimental results we presented to motivate the necessity for a hybrid strategy still hold.

There has also been some work on the notion of converting path expression queries into state machines has been previously proposed in [3,14]. In [14], the authors present the X-Scan operator for evaluating regular path expression queries over streaming XML data. Their work is similar to the ***CA***-FSM presented in this paper, but they handle a wider class of queries, including those with references. In [3], the authors develop indexing and matching mechanisms on a modified finite state machine approach to match XML documents with a large number of user profiles (each expressed as a path expression). The main goal there is to share computation across the evaluation over multiple path expressions. On the other hand, in this paper we are looking at hybrid plans for evaluating a single path expression. An efficient algorithm for processing XPath queries in the presence of navigational access methods only is presented in [13]. This provides an alternate algorithm for the Unnest operator.

In [17], several algorithms were proposed for optimizing branching path expressions in the presence of navigational access methods only. In [24], five algorithms for structural join optimization for XML tree pattern matching queries were presented. In this paper we considered the optimization of path expression queries using both structural joins and navigational access methods.

Recent research studies [1,7,12,20,23] have considered the problem of maintaining summary structures of XML documents to provide statistics information. This work would be useful in estimating the relative cost of the various plans presented in this paper.

XML management systems have been also built on top of either relational [22] or object-oriented [11] systems. Since our system is a native XML database system, our main difference to those approaches is that we do not have to go through the intermediate steps of mapping XML documents to relations or persistent objects and translating queries over the XML documents to the underlying system's query language. An interesting approach is the one in [4] in which the authors employ a hybrid storage mechanism for storing XML documents; they can either store it in flat files, an RDBMS or an OODBMS depending on the XML document structure. Again, our approach is orthogonal to this one. Finally, [21] is a commercial native XML management system; however, there is not enough information about its architecture to date.

Our system is most closely related to [15] which implements a similar system architecture, keeping the same basic distinction between an IR component and an XML data component. Our approach of mixed mode XML query processing would apply to that and other similar systems.

6. CONCLUSIONS AND FUTURE WORK

We have shown that a mixed mode XML query processing system can outperform inverted list filtering and standard query engine navigation techniques when considered separately. Our cost model is accurate enough to choose a quality plan from a large search space.

With our current implementation of ZigZag Join, we only consider single axis paths such as A/B or A[B]. This means that handling a path with N axes requires N ZigZag Join operators. We could consider a more complex structural join operator such as the one presented in [5]. Integrating such an algorithm into our system and extending our cost model and search strategy to explore this larger space of hybrid plans is interesting future work.

Our system is designed to support parallel execution of operators, and thus could benefit from allowing bushy execution plans. We plan to extend the cost model to allow the possibility of choosing a bushy plan. Because the ZigZag join algorithm requires both inputs to be sorted by start number, this will require the optimizer to consider a large number of sort orders for internal nodes in the plans.

Our query workload and cost model are somewhat simple, but they illustrate our key points. Extending these results to include more complex queries and cost models is an area for future research.

Acknowledgement: This work was supported by the National Science Foundation under grant ITR 0086002.

7. REFERENCES

[1] Aboulnaga et al. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. International Conference on Very Large Data Bases, Rome, Italy, September 2001, pp. 591-600.

[2] Al-Khalifa et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In Proc. of ICDE, San Jose, Feb. 2002.

[3] Altinel, Mehmet, Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. VLDB 2000

[4] Barbosa et al. ToX - the Toronto XML Engine. Workshop on Information Integration on the Web 2001: 66-73

[5] Bruno et al. Holistic Twig Joins: Optimal XML Pattern Matching In Proc. of the 2002 ACM SIGMOD International Conference On Management of Data, 2002.

[6] Carey et al. Shoring up Persistent Applications. SIGMOD 1994

[7] Chen et al. Counting Twig matches in a Tree. ICDE 2001

[8] Chien et al. Efficient Structural Joins on Indexed XML Documents. VLDB 2002

[9] dbXML Group. dbXMLCore. Available at http://www.dbxml.org

[10] Dietz, Paul F.. Maintaining order in a linked list. In Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, pages 122-127, San Francisco, California, 5-7 May 1982.

[11] Fegaras, Leonidas, Ramez Elmasri. Query Engines for Web-Accessible XML Data, VLDB 2001

[12] Freire et al. StatiX. Making XML Count. SIGMOD Conference 2002

[13] Gottlob et al. Efficient Algorithms for Processing XPath Queries. VLDB 2002.

[14] Ives et al. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical Report UW-CSE-2000-05-02, University of Washington, 2000.

[15] Jagadish et al. TIMBER: A Native XML Database. The VLDB Journal, Volume 11 Issue 4 (2002) pp 274-291

[16] Li, Quanzhong, Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions, VLDB 2001

[17] McHugh, Jason, Jennifer Widom. Query Optimization for XML. VLDB 1999: 315-326

[18] McHugh et al. Lore: A Database Management System for Semistructured Data. SIGMOD Record 26(3): 54-66 (1997)

[19] Naughton et al. The Niagara Internet Query System. IEEE Data Engineering Bulletin 24(2): 27-33 (2001)

[20] Polyzotis, Neoklis, Minos N. Garofalakis. Structure and Value Synopses for XML Data Graphs. VLDB 2002

[21] Schoning, Harald. Tamino - A DBMS designed for XML. ICDE 2001: 149-154

[22] Shanmugasundaram et al. A General Techniques for Querying XML Documents using a Relational Database System. SIGMOD Record 30(3): 20-26 (2001)

[23] Wu et al. Estimating answer sizes for XML queries. In Proc. of EDBT, Prague, Czech Rep, Mar.2002

[24] Wu et al. Structural Join Order Selection for XML Query Optimization, In Proc. ICDE 2003 (to appear)

[25] Zhang et al. On Supporting Containment Queries in Relational Database Management Systems, SIGMOD Conference, 2001.