NXS: Native XML processing in Sybase RDBMS

Anupam Singh[1], Srikanth Sampath[1], Vadiraja Bhatt[1], Francis Pang[1], Bharat Rane[1], Phil Shaw[1], Gajanan Chinchwadkar[1], Ghazi-Nourdine Benadjaoud[1]

[1]Sybase Inc, One Sybase Drive, Dublin, CA 94568, USA

Email: {firstname.lastname}@sybase.com

Abstract

Sybase's flagship database product, Adaptive Server Enterprise provides Native XML processing (NXS) support. In this paper we present our experience related to query processing and indexing of XML data natively. XML documents are commonly represented as trees. Various tree annotation schemes have been proposed in literature. We use pre-order tree traversal to annotate the tree. This annotation scheme is fundamental part of our approach to native path indexes. We present our industrial strength XPath query engine that exploits the annotation scheme and the path indexes for efficient query processing. One of the key components of the XML engine is an abstract Application Programming Interface(API) named "Path Processor". This API acts as a bridge between query execution engine iterators and access paths. This paper describes the primitives provided by the API on sets of nodes resulting from scans over the XML indexes. The physical query operators in query plans exploit these primitives provided by the path processing API.

1. Introduction

As amount of data online grows rapidly, more and more data is represented in XML. Even though XML data is semi structured, most XML applications require the strengths of a relational DBMS – standard language and industrial strength features such as transactional integrity, concurrency control and high availability. The Sybase XML Engine extends the relational system for efficient storage and querying of XML data. This paper focuses on indexing schemes and query processing that exploits a novel representation of XML data. This makes the RDBMS engine a flexible system that is suitable for managing both relational and XML data.

Recent research has focused on developing a database management system exclusively for XML [2]. Traditionally, Relational Database Management Systems are the preferred repositories for structured data. The main challenges in semi-structured data processing are the navigation of documents based on path expressions and indexing of semi structured data in the absence of traditional schema. An ideal solution would combine the strengths of RDBMS with solutions to the above challenges.

When an XML document is inserted into ASE, it hands over the processing to the XML Engine. The XML document is parsed by the XML Storage Engine as part of the SQL *insert* or *update* statement and transformed into a binary object containing the indexes and stored in an *image* column of ASE. The XML document is queried as part of the SQL *select* statement. ASE 12.5 supports XQL and ASE 12.5.1 supports a subset of XPATH as the XML query language of choice.

Section 2 describes the XML Data Model used by NXS. Section 3 gives an overview of the System Architecture. The XML Storage Engine including the Path Processor is described in Section 4. The XML Query Engine is described in Section 5.

The XML document fragment in Figure 1 is used to illustrate the concepts going forward. A bookstore contains books having a title and information about the authors.

```
<bookstore>
<book>
        <title> Trenton…</title>
        <author>
                <first-name>Mary</first-name>
                <last-name>Bob</last-name>
        </author>

</book>
<book>
        <title>National…</title>
        <author>
                <first-name>Joe</first-name>
                <last-name>Bob</last-name>
        </author>
</book>
```

Figure 1: Sample XML Fragment

2. SYBASE XML Data Model

2.1 XML Document Representation

We represent an XML Document as a rooted, ordered, labeled tree. There is one node, the "root" that is distinguished from other nodes in the tree. The children of a node are ordered, in the sense that there is a first child, second child etc. All XML constructs such as elements, attributes, processing instructions, and comments are represented as nodes in the tree. The type of a node is defined by the XML construct that the node represents. Thus, elements and attributes are represented as "element" and "attribute" nodes labeled with their names. Text fragments inside an element are represented as distinct leaf nodes with a special label '-value'. Only leaf nodes have values associated with them.

Consider an XML document represented as a tree as shown in Figure 2. The depth or level of a node is the number of nodes encountered during a traversal to this node from the root. The path to a node is the set of labels encountered during a traversal to this node from the root. The length of a path is the number of labels in the path.

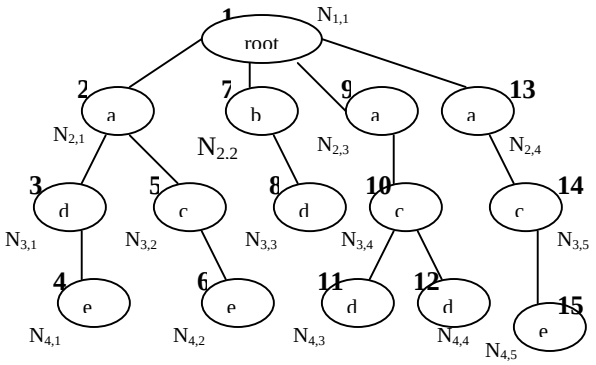

Figure 2: XML Data Model: Document as a TREE

Any node 'm' in the unique path from the root to a node 'n' is an ancestor to node 'n'. If 'm' is an ancestor of 'n', then 'n' is a descendant of 'm'. Every node is identified with an identifier $N_{i,j}$ where 'i' is the depth of the node and 'j' is a number increasing from left to right at a given level. For example, in Figure 2, Node $N_{4,2}$ is at depth '4' and conforms to the path /-root/a/c/e.

2.2 Ancestor-Descendant Relationship

An obvious yet difficult to determine relationship in a tree is the ancestor-descendant relationship. This section defines the problem and considers possible solutions.

Problem: Let $\mathbf{S_1}$ be the set of all nodes that conform to the path P_1:/a/b/…/k at level 'X'. Let $\mathbf{S_2}$ be the set of all nodes that conform to the path P_2:/a/b/…/k/…/n at level 'Y'. Note that the path P_1 is a prefix to the path P_2 and also Y > X. Find the set $\mathbf{S_1}'$ which is a subset of $\mathbf{S_1}$ such that $\forall\ \mathbf{s_1}' \in \mathbf{S_1}$, $\exists\ \mathbf{s_2} \in \mathbf{S_2}$ and a path from $\mathbf{s_1}'$ to $\mathbf{s_2}$. In other words, find the subset of nodes from set $\mathbf{S_1}$ such that they are ancestors to some node in $\mathbf{S_2}$.

Example 1: Consider the path P_1:/-root/a, and path P_2:/-root/a/c/e for the XML document represented in Figure 2. The problem translates to finding the subset $\mathbf{S_1}'$ from the set $\mathbf{S_1}$: $\{N_{2,1}, N_{2,3}, N_{2,4}\}$ that contain the ancestors to elements in set $\mathbf{S_2}$: $\{N_{4,2}, N_{4,5}\}$.

2.2.1 **Solution1: Tree Traversal Approach.** In this approach, the tree is traversed either top-down or bottom-up. In the top-down traversal, we start at each node in set S_1 and traverse the sub-tree rooted at the node conforming to the suffix of path P_2 after P_1, until a node is reached at level 'Y'. If this node is in the set S_2 then add the node from S_1 to S_1'. In the bottom-up traversal, we start at a node in S_2 and follow the parent (provided parent pointers exist) until a node at the level 'X' is reached. Both approaches have the drawback in that tree traversals need to be performed and intermediate nodes need to be visited.

2.2.2 Solution 2: Pre-Order Annotation Approach. We perform a pre-order traversal of the tree and every node is annotated with a monotonically increasing sequence of natural numbers. This unique identifier of a node is referred to as the Node Identifier or NID. The NIDs are shown as bold numbers in Figure 2. We make the following observations about the annotated tree:

1. Node $N_{i,j}$ will be visited after visiting every node in the sub-tree rooted at node $N_{i,(j-1)}$
2. NID($N_{i,j}$) is greater than the NID of every node in the sub-tree rooted at $N_{i,(j-1)}$
3. NID of a node in the sub-tree of $N_{i,j}$ will be greater than the NID of $N_{i,j}$

From the above assertions, it follows that NID of a descendant of node $N_{i,j}$ will have an NID greater than NID of $N_{i,j}$ and less than NID of $N_{i,(j+1)}$.

Given, $\mathbf{S_1}$: {…,$N_{x,(j-1)}$, $N_{x,j}$, $N_{x,(j+1)}$, …} $\mathbf{S_2}$: {…$N_{y,k}$, …},

And NID($N_{x,(j-1)}$) < NID($N_{y,k}$) < NID($N_{x,j}$)

We can conclusively determine that $N_{x,(j-1)}$ is an ancestor to $N_{y,k}$

If NS_1 and NS_2 represent the set of NIDs for paths P_1 and P_2, then the algorithm in Figure 3 can be used to arrive at the subset $\mathbf{S_1}'$. In Example 1, S_1:{2, 9, 13} S_2:{6, 15 }. Executing the above algorithm, $\mathbf{S_1}'$:{2, 13}. The algorithm is better than solution 1 as:

1. It eliminates tree traversals
2. The set $\mathbf{S_1}'$ is arrived at by a single pass through the sets NS_1 and NS_2.

```
i=0; j=0;
while (NS2(i) exists)
{
        if (NS1(j) < NS2(i) < NS1(j+1))
                add NS1(j) to S1'; i++
        else
                j ++;
}
```

Figure 3: Algorithm for Ancestor-Descendant Relationship

However, we make two assumptions. First, given paths P_1 and P_2, the sets NS_1 and NS_2 can be obtained. This is our motivation for the Path Index in the Section 4.1. Second, the annotation of the tree is not expensive. This is the motivation for annotation and index generation during XML parsing as explained in Section 4.1.

3. System Architecture

Figure 4 shows the system architecture for the XML Engine running within Sybase Adaptive Server Enterprise.

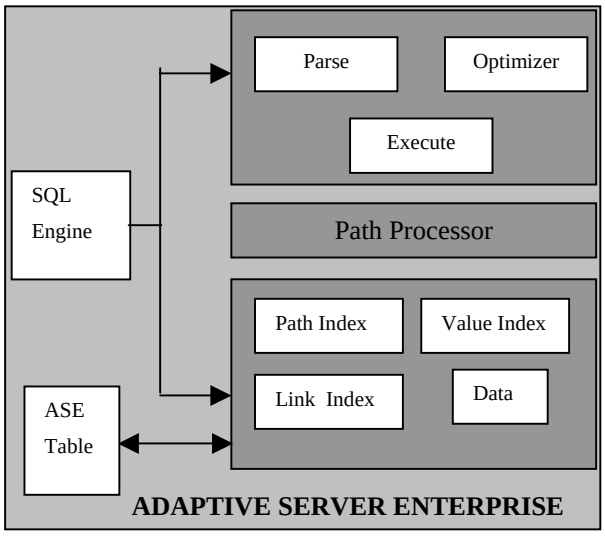

Figure 4: System Architecture

The XML document fragment in Figure 1 is represented as a tree and is annotated in the pre-ordered approach as described in the Section 2.2.2. In ASE 12.5.1, the following syntax is used to annotate and transform the XML document including generation of indexes as described in Section 4.

insert xmltab select xmlparse(xmltextcol) from rawxmltab

An XML query can be translated to actions against the tree such as extraction of all nodes that conform to a given path, qualification of ancestors based on characteristics or existence of descendants, qualification of sub-trees based on another sub-tree and retrieval of nodes at arbitrary depths with unspecified paths.

Example 2: The following XPATH query will be used as a motivating example in subsequent sections. To retrieve the title of books with author's whose first-name is 'Joe', following XPath extension to SQL will be used

select xmlextract("/bookstore/book[author/first-name = 'Joe']/title", xmlcol) from xmltab

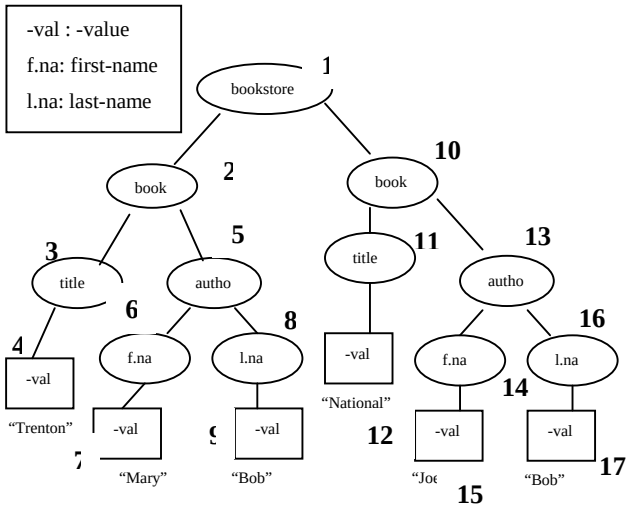

Figure 5: Annotated Sample XML Document

This is optimized and translated to a plan consisting of the following path requests as explained in Section 5

1. /bookstore/book
2. /bookstore/book/author/first-name = 'Joe'
3. /bookstore/book/title

Path requests that qualify nodes based only on paths such as (1) and (3) are called *"simple path"* requests. Path requests that qualify nodes based on path, value and a relational operator such as (2) are called *"predicated path"* requests. The plan is executed against the services provided by the Path Processing layer (Section 4.2). The

result is a set of nodes in the tree and an XML document fragment is generated as explained in Section 5.4.

4. XML Store

The store engine has two basic functionalities:

1. Parse Time Services - To annotate and transform an XML document including generation of indexes for faster access
2. Run Time Services - To resolve path requests from the query execution layer and regenerate XML document fragments

4.1 Parse Time Services

During the parsing of an XML Document, we identify every XML construct with a unique identifier, the node identifier (NID) and generate the Path Index, Value Index and the Link Index.

4.1.1 Path Index. The algorithms in Section 2.2.2 relied on the assumption that given a path, the set of NIDs can be obtained in constant time. We achieve this through the Path Index. For every unique path in an XML document, we store the set of NIDs that conform to the path in the document order of occurrence. An XML document can have many paths, but the set of unique paths are finite. Table 1 shows sample Path Index entries for the document in Figure 1.

Table 1: Path Index

Path	NID
/bookstore/book	2,10
/bookstore/book/author	5,13
/bookstore/book/title	3,11

4.1.2 Value Index. Predicated path requests do not need the complete set of NIDs that conform to a path, but only a subset of them that match a given "value" as defined by the "relational operator" in XPATH. For every unique path that terminates in a leaf (a leaf-path), the (value, NID) pairs are stored.

Table 2: Value Index

Leaf Path	NID
/bookstore/book/author/first-name/-value	(Joe, 15), (Mary, 7)
/bookstore/book/author/last-name/-value	(Bob, 9,17)
/bookstore/book/title/-value	(National,12), (Trenton, 4)

Note that only the leaf nodes in an XML tree have values associated with them and hence only leaf-paths have an entry in the Value Index. The NIDs stored in the value index correspond to the value nodes and the nodes that contain them (the parent nodes) are obtained using the Link Index (Section 4.1.3). Table 2 shows sample Value Index entries for the XML Document in Figure 1.

4.1.3 Link Index. The Link Index maintains the parent-child relationship in the XML Tree. The Link Index is used to obtain the parent in predicated path requests, result generation and XPath constructs such as subscripts. The Link Index consists of two components. First, for every node, the NID of its parent is stored. Second, for every node, the NIDs of the children are maintained. Table 3 shows a sample of the Link Index for the document in Figure 1.

Table 3: Link Index

NID	PARENT	CHILDREN
2	1	3,5
3	2	4

4.1.4 **Data.** Data value of each node such as the node type, name, NID, value (if any) are stored in the data portion of the transformed document. ASE provides fast access structures that enable to randomly seek to any portion of the transformed document. The indexes are serialized along with the data and are stored in an *image* column of ASE.

4.2 Run Time Services – Path Processor

During query execution, the query execution layer interacts with the store layer through a well-defined API called the Path Processor. This API acts as a firewall and provides the ability for the XML engine to seamlessly adapt to a new path based query language or a new storage engine. The result of a service request is always a set of nodes. The Path Processor defines the set of operations that are permissible on this set of nodes. The result of these operations (if any) is another set of nodes. The Path Processor does not provide visibility to the contents of the set, but provides an iterative interface over the set to the query execution layer. The usage of the services provided by this layer is discussed in Section 5. It is important to note that all path processing services are over sets of

integers. Thus, requests from the query processing layer are degenerated to integer operations.

The services provided are:

1. Simple Path Requests – Given a simple path such as /bookstore/book, the Path Processor consults the Path Index and returns the set of nodes that conform to the path. With the path index, given a path, the set of NIDs that conform to the path are retrieved in constant or logarithmic time, as the number of unique paths in a document is finite.
2. Predicated Path Requests – Given a predicated path such as /bookstore/book/author/first-name = "Joe", the Path Processor consults the Value Index and Link Index and returns the set of nodes that conform to the predicated path. Since the number of unique leaf paths is finite, given a leaf path, its entry in the Value Index can be arrived at in constant or logarithmic time. Given a value and a comparison operator, the subset of NIDs that match the value can be arrived at in logarithmic time. For example, given a path /bookstore/book/author/first-name, value "Joe" and relational operator "=" we retrieve the value node 15 from the Value Index. We consult the Link Index and obtain 14 as the parent node that has a value node 15, and add 14 to the set of qualified nodes.
3. Subscripts – Given a node and an index (or a range), the Path Processor consults the Link Index and returns a set that contains the child(ren) of the node at that index(range).
4. Identity Comparison – Given two nodes from the set of possible ancestors and a node from the set of descendants, the Path Processor qualifies the ancestors and the usage is explained in Section 5.3
5. XML Materialization – Given a set of nodes, the Path Processor retrieves the XML document fragment rooted at the nodes as explained in Section 5.4.

5. Query Processor

Query processing in the XML engine follows the typical steps of parsing, query optimization and query execution. The XPath query parser generates a parse tree. The Query Optimizer translates the parse tree into a physical plan.

5.1 Query Optimizer

The goal of the query optimizer is to generate plans that avoid top-down tree traversals and text based XML fragment processing. This is achieved by translating XPath language primitives into directives to the Path Processing layer. This mapping between language level operators and path processing layer directives is strictly rule based.

The optimization is driven by exploitation of capabilities provided by the Path Processing layer. We will look at the rules based on the query "/bookstore/book/author/first-name = 'Joe']/title" introduced in Example 2. The physical plan is shown in Figure 6.

5.1.1 **Path Scans.** The parse tree is decomposed into three 'undivided' paths with two simple paths, "/bookstore/book" and '/bookstore/book/title', and a third predicated path, "/bookstore/book/author/first-name = 'Joe'". The simple paths are translated to Path Scan (SCAN) operators, OP5, OP7 and OP8. The predicated path is translated to a Predicated Scan operator (PSCAN), OP6.

5.1.2 **Ancestor-Descendant relationship.** The next step is to qualify ancestors based on properties of their descendants. The ancestor-descendant property of the data model described in Section 2.2.2 is abstracted in a physical operator called 'XQUAD' (XML Qualification of Ancestor based on Descendants). The optimizer creates an 'XQUAD' operator which qualifies a 'book' node on the left hand side only if the node has a 'first-name' descendant node on the right hand side with a value "Joe". OP3 is introduced into the plan to retrieve nodes in OP5 that are ancestors to the nodes in OP6. Similarly, an XQUAD, OP4, is generated for the two path scans, OP7 and OP8.

The optimizer creates an 'XQUAD' operator which qualifies a 'book' node on the left hand side only if the node has a 'first-name' descendant node on the right hand side with a value "Joe". OP3 is introduced into the plan to retrieve nodes in OP5 that are ancestors to the nodes in OP6. Similarly, an XQUAD, OP4, is generated for the two path scans, OP7 and OP8.

5.1.3 Set Operations. Qualification of a sub-tree based on another sub-tree can be transformed to qualification of common ancestors. Each XQUAD operator groups ancestors with a set of descendants. Therefore, we intersect the set of descendants on common ancestors and project relevant descendants. The intersect operator, OP2, thus qualifies only the 'book' nodes qualified by both sub-trees and projects 'title' nodes returned by the XQUAD operator OP4.

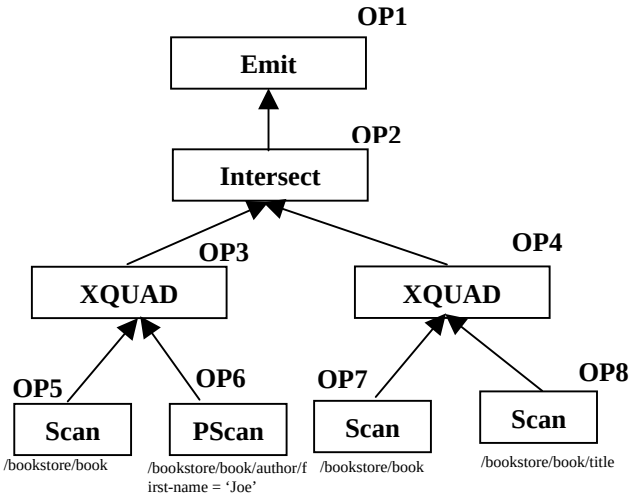

Figure 6: Optimized Plan

5.1.4 **Result Generation.** The EMIT operator, OP1, is generated to create the textual representation of the projected nodes.

5.2 Wildcards

An important requirement in path based query languages is retrieval of nodes at arbitrary depths. Structural summaries like DataGuides [3] can be used to expand regular path expressions in queries. The Query Engine utilizes the Path Index as a DataGuide to achieve compile time expansion of paths for steps containing descendant axes and wildcards. For example, for "/bookstore//title", we use the path index to get set of paths that have the prefix 'bookstore' and suffix 'title'. Another example is '/bookstore/*/title where we use the path index to obtain the set of paths for all children of 'bookstore' which have a child 'title'.

5.3 Query Execution

As has been the accepted design for relational databases, all the physical operators are designed as recursive iterators over data items based on the Volcano Engine[1]. Unlike iterators in relational database these iterators act on sets of nodes rather than sets of tuples. Each iterator acts in a *'single-node-at-a-time'* environment. The XML Query Execution Engine has many iterators such as Union, Intersect, Scan, Subscript and XQUAD.

We describe two important operators here. Figure 7 depicts a sample of the flow of nodes in the plan for Example 2. The arrows, ↑, between operators show the flow of nodes. The NODE Ids such as {2} refer to the NIDs depicted in Figure 5.

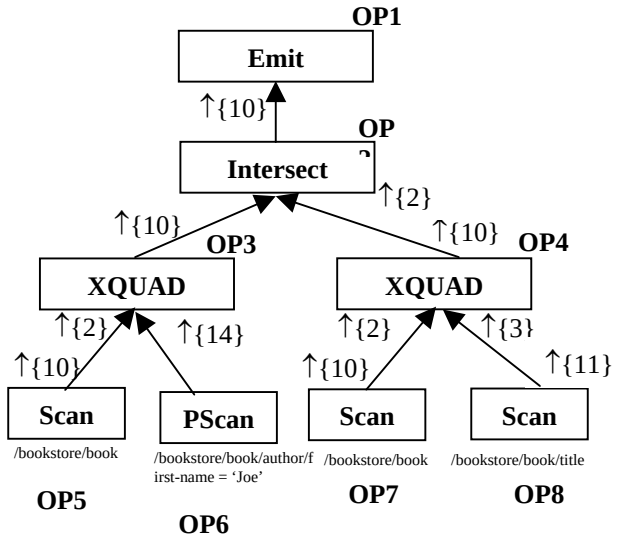

Figure 7: Plan Execution

5.3.1 **SCAN.** A SCAN iterator translates to iteration over the scans in the path processing layer. Path scans OP5 and OP6 generate NID {2} and NID {14} respectively. A *next()* call on OP5 generates NID {10}.

5.3.2 **XQUAD.** The XQUAD operator uses the property described in Section 2.2.2 to impose ancestor-descendant relationships. Node identity comparisons are pushed to the path processing layer. The XQUAD iterator implements the algorithm described in Figure 3.
In OP3, as (2 < 14 < 10) is FALSE, it is determined that {14} cannot be a descendant of {2} and {2} is rejected by OP3. Doing the same exercise for {10} and {14} makes {14} a descendant of {10} and hence OP3 qualifies {10}. A similar sequence of operations qualifies {2} and {10} in OP4. The result of intersection in OP2 is {10}. OP1 materializes the XML representation of {11}, a descendant of {10}.

5.4 Result Generation

Unlike SQL systems, there is a cost of 'creating' the result in an XML Query from the qualified set of nodes. The Path Processor provides services to project fragments of the XML documents rooted at these nodes. Our solution exploits the annotation of the XML tree and the Link Index. Consider that the sub-tree rooted at a node $N_{i,j}$ needs to be projected. We make the following assertions based on the Sybase XML Data Model

Parent NID($N_{i,j}$) < Parent NID(every element in the sub-tree of $N_{i,j}$)

Parent NID($N_{i,(j+1)}$) < Parent NID(every element in the sub-tree of $N_{i,j}$)

Starting with $N_{i,j}$, the data is retrieved and the Node Id incremented iteratively as long as the Parent NID of the node is greater than the Parent NID of $N_{i,j}$ as determined by the Link Index. Note that we do not retrieve the children of any node.

Example 3: Consider the projection of the sub-tree rooted at node id 2 in Figure 5. We note 1, the parent node id of starting node as the initial parent node id. Starting with node 2, we continually increment the current node id. For each current node, we retrieve the data for the node, format it and append to the result XML fragment, as long as the parent node id of the node is greater than the initial parent node id of 1. Thus, we terminate at node 10.

6. Performance

Consider the execution times (in milliseconds) for the queries in Table 4 for document sizes 10K and 4M. Each query is based on the XMARK[4] DTD and has a distinct property.

Table 4: Performance results

	QUERY	NATURE	10 K	4M
1	/site/regions/africa/item/@id	Small Result	9	90
2	/site/regions//item	Large Result	23	6836
3	/site/regions/africa/item[@id= ‘item4’]/name	Narrow Predicate	9	33
4	/site/regions/africa/item[@id >= ‘item1’ and @id <=‘item20’]/name	Wide Predicate	10	66
5	/site/closed_auctions/closed_auction[annotation/description/parlist/listitem/text/emph/keyword/text()]/seller/@person	Long Paths	10	163

Query 1 demonstrates that the execution time (increases 9 times) is not linearly dependent on the size of the document (increases 400 times). Queries 3 and 4 demonstrate the power of the XML Engine with predicated queries. The fact that the execution time is independent of the path length is exhibited in Query 5. Result generation is dependent on the number of nodes in the result set and this fact is borne out in execution times for Query 2.

7. Acknowledgments

Special thanks to Raghavan Eachampadi for his contribution toward the XML storage architecture. For their contributions to system implementation many thanks to (alphabetically) Kannan Ananthanarayanan, Vinod Chandran, Yan Chen, Xun Cheng, Sudipto Chowdhuri, Nick Lyons, Sethu Meenakshisundaram, Kartik Mudaliar, Moidin Rehmatullah and Tiger Whittemore. Finally, we are grateful to Jennifer Widom at Stanford University for many fruitful discussions.

8. References

[1] G. Graefe**.** Query evaluation techniques for large databases. ACM Computing Surveys. 25(2): 73-170,

[2] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data**.** SIGMOD Record, September 1997..

[3] J. McHugh and J. Widom. Compile-Time Path Expansion in Lore. Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Israel, Jan 1999.

[4] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I Manolescu, M. J. Carey, R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.