IADIS International Journal on Computer Science and Information Systems
Vol. 1, No. 1, pp. 84-100
ISSN: 1646-3692

SERVICE BASED XML QUERY EXECUTION ENHANCER

Weiyi Ho
School of Computer Science & IT,
Nottingham University, Nottingham NG8 1BB, UK

Li Bai
School of Computer Science & IT,
Nottingham University, Nottingham NG8 1BB, UK

ABSTRACT

Extensible Markup Language (XML) is a language for storing and exchanging structured and semi-structured data in a flexible format. As previous sources support public XML export, increasing amounts of public and private data are described in XML. Due to the flexible characteristics of XML it is today employed for the representation of data in virtually all areas of computing, and particularly in web-based applications. One of the most important tasks in XML document processing is querying and navigating, that is the extraction of parts of a document that match a structural condition and are in a specific context. However, due to the nature of current file storage and the tree-like view of XML documents, XML file accessing is in a sequential manner and the navigation is often costly. Great challenges arise from the demands of integrating and manipulating XML data from different sources. In this paper, we address the fundamental issues and develop a comprehensive solution with regard to the enhancement of XML structural query execution. We present this enhancing process in detail, and further provide the operation examples to demonstrate how they would work. We also conduct a series of experiments based on our proposed approach and report our findings.

KEYWORDS

Semi-structured Data, Query Execution, Service Orientation, J2EE.

1. INTRODUCTION

With the rapid development of Internet technology large volumes of data in the form of electronic documents have been created. This includes, in particular, semi-structured textual information such as electronic documents, programs, log files, online text, transaction records, patent information, literature citations, business profiles, and emails. The characteristic of this data varies greatly in structure. XML provides the necessary flexibility desired for handling data with such varying structures. As XML is becoming the principal medium for data exchange over the Web and for information integration in general, ever increasing amounts of public and private data are described in XML. XML data is usually defined in a tree or graph based, self-describing object instance model (Deutsch et al., 1999). It is also known as an easy-to-write and easy-to-parse language which offers a better way to exchange data between varieties of applications on the Internet (Bourret, 2004). The extensible nature of XML provides flexibility in describing the variable structure of data. However, semi-structured data is incompatible with the flat structure of relational database tables, and therefore the growth of XML data requires new and complex storage and manipulation techniques.

Data analysis in the relational world is relatively complicated since the data is derived from business processes or human activities. However, for the purposes of data integration and data exchange, the data originates mostly from the existing electronic data. By considering the characteristics of data sources, the data which is available electronically can be divided into two broad categories. The first, data centric, is mainly for machine consumption, such as the data in relational databases. The second, document centric, is for human consumption, such as the data located in a file system. Moreover, from the enterprise perspective, a variety of information management systems have been widely adopted in daily business activities. However, each of the systems has its own data structure and access methods which results in data heterogeneity. In addition, enormous volumes of electronic data are located on the Web, which can be seen as a world wide file system. Anyone with access to the Internet can use this storage system to publish and access electronic documents. It is a similar situation with Intranet and Extranet platforms for business applications.

One possible solution for facilitating the task of manipulating the data form heterogeneous sources as mentioned above is to utilize XML as a medium to bridge the server and the client. Consequently, one common interface is used to access the data from different sources. It breaks down the barriers between different computing platforms, development environments and communications networks, allowing organizations to work together electronically without the expense and delay of agreeing on semantics, schema, interfaces, and other application integration. We use J2EE XML Web Services framework to demonstrate this idea (Sun Microsystems, 2003). XML data is the principal medium in this architecture. We propose a more effective and efficient manipulating technique in the Service based fashion in order to fill in the gap between user requests and data sources. The position of our work and the complete architecture can be found in Figure 1.

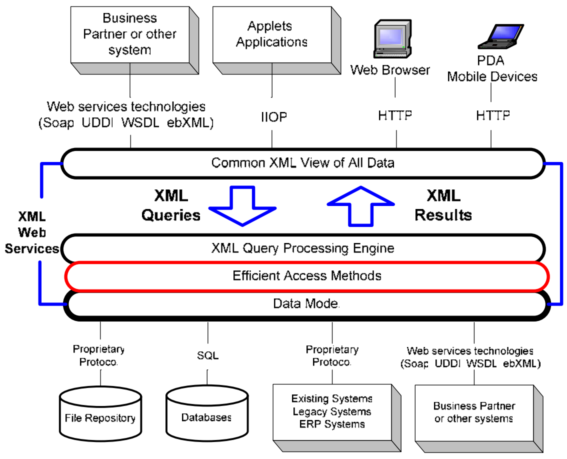
Business Partner or other system
Applets Applications
Web Browser
PDA Mobile Devices
Web services technologies (Soap UDDI WSDL ebXML)
IIOP
HTTP
HTTP
Common XML View of All Data
XML Queries
XML Results
XML Web Services
XML Query Processing Engine
Efficient Access Methods
Data Mode
Proprietary Protoco
SQL
Proprietary Protoco
Web services technologies (Soap UDDI WSDL ebXML)
File Repository
Databases
Existing Systems Legacy Systems ERP Systems
Business Partner or other systems

Figure 1. XML Web Services Architecture

The navigation of tree-natured XML data is proven to be costly. Evaluation of the regular path expressions involves the exploration of the descendant of a given set of nodes. Too many nodes have to be visited unnecessarily due to the blindness of a subtree (descendant) to a node (Sangwon Park & Hyoung-Joo Kim, 2002). Accessing through tree structure requires recursive computation, for a complex query it is most likely to incur redundant tree travelling which could greatly downgrade overall query execution performance. Accordingly, accessibility plays an important role for handling node-to-node relationships in XML related framework. Moreover, the textual aspect of XML data is as important as the structural parts. A full text search in tree structure, such as DOM, is very difficult, since the textual data of the XML document is embedded in the tree nodes. Further, querying among large volumes of XML documents is quite often the case in a document-centric situation. Although XML related fields have been intensively studied in recent years by the database research community, little research has been done on path evaluation in large volume XML documents. In this paper, we propose two efficient approaches based on the prevention of unnecessary visits to, first, the documents and, second, the subtrees. By doing so, we can enhance query execution performance significantly.

The rest of the paper is organized as follows. Section 2 discusses the background in XML and the observations that led us to the introduction of the new methods. The proposed methods are described in detail in Section 3. A thorough operation example is presented in Section 4. In Section 5, we report the findings of the experimental results. Finally, section 6 contains some concluding remarks and directions for future work.

2. BACKGROUND STUDIES

2.1 Data Model and Queries

XML is a general markup language proposed by the World Wide Web Consortium (W3C). Due to its inherent flexibility, much of the data encoded in XML will be semi-structured. This indicates that the data may be irregular or incomplete, and that its structure is highly flexible. Semi-structured data is generally represented as a graph or a tree. The Document Object Model (DOM) is a standard interface of XML data, whose structure is a tree, which is the data model used in this paper (W3C, 2004). Figure 2 is an example of an XML document, and its DOM structure, also known as XML data graph, is represented as a tree in Figure 3. We will use this example XML document throughout this paper.

Queries in XML query languages like XML-QL, Quilt, and XQuery are based on tree patterns for matching relevant portions of data. In this paper we focus on evaluating regular path expressions. The wild card operators such as “*” and “?” are allowed in path query. When a query is being processed, a scan operator is provided to dig out nodes which match the given path expression. Normally this process needs to access the entire data to fetch a node. Therefore, the number of nodes visited decides the query evaluating performance. The aim of this paper is to minimize the number of nodes fetches in order to reduce the cost of evaluating the queries (Daniela Florescu & Donald Kossman, 1999;Jason McHugh & Jennifer Widom, 1999;Jayavel Shanmugasundaram et al., 2001) .

```
<?xml version="1.0" ?>
        <menu>
          <food category="breakfast">
                    <name>Full English Breakfast</name>
                    <price>6.95</price>
                    <description>Two eggs, bacon or sausage, toast…
             </description>
                    <calories>970</calories>
             <chef id="001">
                <first>Mick</first>
                <family>Burton</family>
                <email>mick@burton.com</email>
             </chef>
          </food>
          <food>
                    <name>Old-Fashioned Breakfast</name>
                    <price>7.03</price>
                    <description>Eight eggs, complete pig, loaf of bread…
             </description>
                    <calories>1150</calories>
                    <chef>
                <first>Jack</first>
                <middle>A.</middle>
                <family>Smith</family>
                <phone>123-4567</phone>
             </chef>
          </food>
          <drink category="Hot">
                 <name>Tea</name>
                    <price>2.50</price>
                    <description>A pot of Tea</description>
                    <calories>100</calories>
          </drink>
        </menu>
```

Figure 2. Example XML data

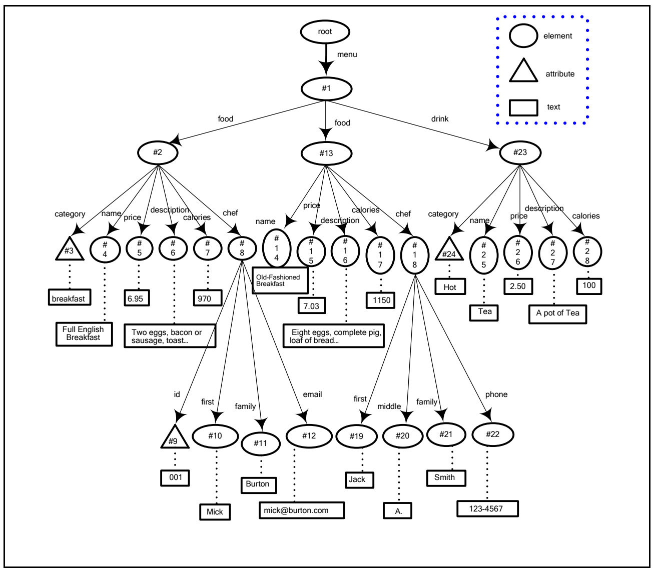

Figure 3. XML DOM model tree representation

The signature methods described in (Haifeng Jiang et al., 2002;Man-Kwan Shan & uh-Yin Lee, 1998;Yangjun Chen, 2002) have particularly focused on text retrieval and object-oriented methodology. Signature files are collections of signatures and are widely accepted due to low space overheads and reduced update costs (Chris Faloutsos, 1985).

Signature, denoted by *S*, is a bit string of *F* bits, where *F* is the signature length. A signature is basically an abstraction of information which is used to indicate the presence of an individual in sets. A hash function is provided to encode each element of a specific set into a signature, which will set exactly *m* out of *F* bits to *1*. The value of m is called weight of the element signature. The set bits are uniformly distributed in the range of *[1.. F],* since the hash function *H* is assumed to have ideal characteristics. Therefore, the probability of a bit position in an element signature being set to one is equal to *m/F*. Finally, the set signature is generated by applying the superimposed coding technique on all element signatures, i.e. all positions are superimposed by a bitwise OR-operation to generate the block signature. An example is shown in Table 1 below:

Table 1. Example Block Signature Generated by OR-operation

Menu	0000 0100 1000
Food	0000 0010 0010
Drink	0000 1000 0010
Block Signature (S_B)	0000 1110 1010

The inclusion of a specific element can be determined by a bitwise AND-operation. If $H_{\text{"food"}} = H_{\text{"food"}} \cap S_B$, then there is a high possibility that “food” exists in the block. However, if we test $H_{\text{"category"}}$ = “0001 0000 1000” on the block signature, the result of $H_{\text{"category"}} \cap S_B$ is not $H_{\text{"category"}}$. This indicates that there is no element “category” in that block.

Since signatures are abstractions of the original represented information, they introduce information loss. False drop might happen when unqualified element passed the bitwise AND-operation test. Although, increasing the length of signature or reducing the number of elements in a block can lower the probability of false drop, it also incurs a disk space overhead.

In this paper, we apply the signature approach and propose new access methods in order to offer a better performance in retrieving XML documents. The signature of each node is formed by first hashing each value in the block into a bit string and then superimposing all bit strings generated from the block into the block signature. The concept of “block” here is actually the subtree of the XML DOM tree. The details will be discussed in Section 4.

We define some notations for signature description. Let the hash value of the name of a node *i* be *H(i),* and the node signature be *NS(i).* The *NS(i)* is the ORing of all the hash values of node *i*’s descendant nodes. That is, the hash value is propagated to its parent node. Then we can estimate the existence of a certain name *x* in the subtree of node *i* by comparison of $H(x) \cap NS(i)$. If $H(x) = H(x) \cap NS(i)$ then there may be the name *x* in the subtree of node *i*. Otherwise, if $H(x) \neq H(x) \cap NS(i)$, then we can be assured that the name *x* does not exist in the subtree.

3. TWO-STEP XML FILTERING APPROACH

In this paper we propose a two-step signature-based navigation method: the *Level Signature* (LS) and the *Sequential Signature table* (SST). Firstly, we assume an XML document is represented as a DOM tree, an example of which can be found in Figure 3, and there is hash function *H* for creating the hash value. The label path contains the names of the element or attribute in the DOM tree. Therefore only element and attribute nodes are involved in making the signature. Figure 4 depicted our approach.

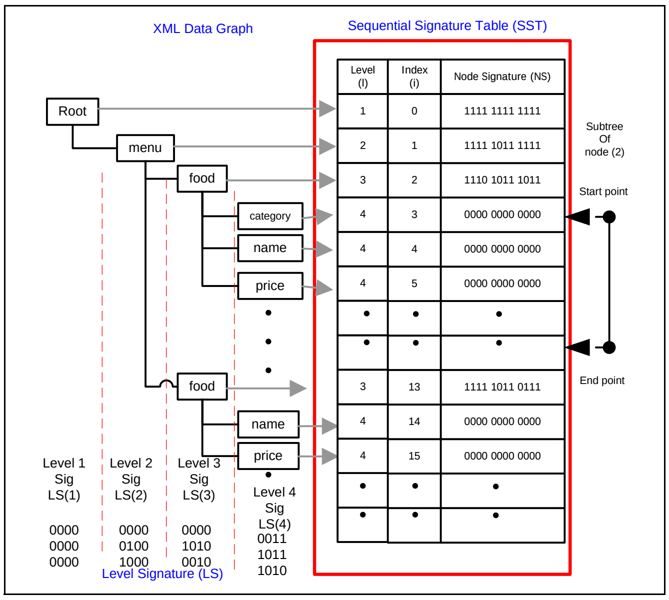

Figure 4. LS and SST based on the example data

3.1 First Step - Level Signature (LS)

The level signature is the OR-ed signature of the nodes that reside in the same level of the DOM tree. Moreover, the level signature path (LSP) is the path of level signature arranged according to the level of the DOM tree.

Definition 1. (Level Signature): We assume the number of nodes in document *D* is *n*. The level signature of the *l*th level of document *D* is denoted by $LS_D\ (l)$.

$$LS_D\ (l) = OR\ Operation(\ if\ H(i)\ in\ level\ l\ )\ \ where\ i = 0\ ..\ n\text{-}1.$$

In addition, the level signature path of an XML document *D*, denoted by LSP_D, where the height of document *D* is *h*.

$$LSP_D = LS(1)\ /\ LS(2)\ /\ \dots\ /\ LS(h)$$

3.2 Second Step - Sequential Signature Table (SST)

The *sequential signature table (SST)* includes three fields, *level (l)*, *index (i)* and *node signature (NS)*, respectively. To begin with, *level (l)* represents the level of a node in the XML DOM model, which can be easily derived from the path expression of a node. For example, the path expression of the "food" element is "#/menu#/food", so we know it is located in level 3 (root in level 1). Furthermore, the order of *sequential signature table* is basically obtained from depth first traversal of the DOM tree, and corresponds to index *i,* where *i = (0 ... n-1)* in a DOM tree with *n* nodes in total. For a better understanding of the order, reader can refer to Figure 4.

$$SST(Level,\ Index,\ NodeSignature)$$

Definition 2. (Node Signature): The node signature of *i*th node of document *D*, denoted by $NS_D(i)$ and *i* represents the index of node. The hash value of *i*th node is denoted by *H(i)*. We assume the number of descendant of *i*th node is *m*. (Here we use "U" to denote the bitwise OR operator.)

$$NS_D(i) = H(i+1)\,U\,H(i+2)\,U \ldots U\,H(i+m)$$

As mentioned earlier in this section, the index in SST represents not only the order, but also provides a direct mapping of each node in the DOM tree and follows depth first traversal. In building the SST tables, the idea of using index plays an important role in keeping tree structure and node signatures in line. From definition 2, the node signature (NS) is derived from ORing all the hash values of the nodes in the subtree (all descendants). The SST is in a sequential form of tree structure, and it is also a depth first traversal. Because of that there is an interesting feature in the SST table, which physically clustered the subtree together on a sequential basis. This property offers an efficient way to skip an unqualified subtree. We then show this concept in Definition 3 below.

Definition 3. (Subtree of a node): The *i*th node in SST denoted as *node(i),* and subtree of *i*th node as *ST(i).* We assume the number of descendant of *node(i)* is *m*.

$$ST(i) = node(i+1),\ node(i+2),\ \ldots,\ node(i+m)$$

Since the subtree is serial, the start position of a subtree will be the next node of *node(i)* which is *node(i+1)* shown in the above equation. Then we need to know the position at which that subtree ends. Level will be an indicator for the sign of end, as all the nodes in a subtree should have a greater level than the parent node. Therefore, if the level of a node is equal or less than the level of *node(i),* that indicates the end of the subtree.

3.3 XPath Accessibility in LS/SST Approach

In this section, we will focus on the accessibility aspects of the XPath language in our proposed approach namely LS/SST. The XPath language is used for addressing nodes in a document or matching sets of nodes by their relationships to a context node. An XPath navigation expression is a sequence of navigation steps separated by the "/" symbol. The expressions used to select nodes are called location paths. The axis describes the tree relationship to the context node and defines how to navigate the document tree.

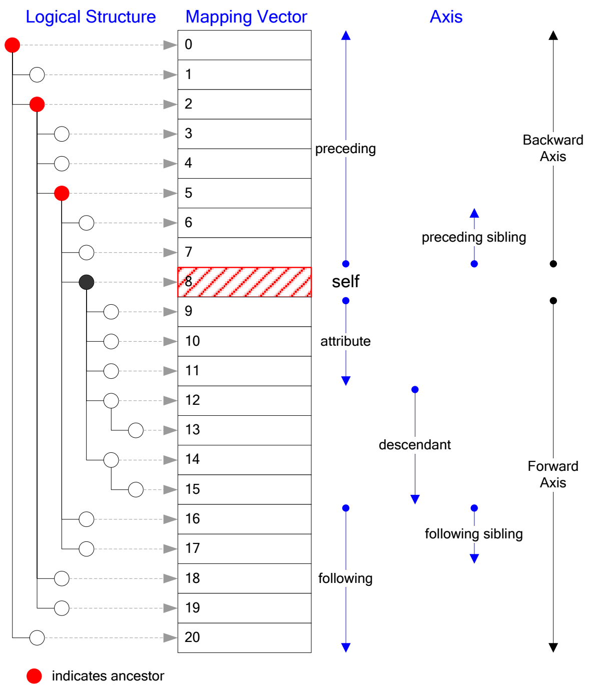

Figure 5. XPath Axis in LS/SST Model

A navigation expression returns a list of nodes. The order is defined by the document order for forward axis or the reverse document order for backward axis. The possible value for AxisSpecifier are child, descendent, parent, ancestor, following-sibling, preceding-sibling, following, preceding, self, descendent-or-self, and ancestor-or-self etc. Figure 5 clearly demonstrates the connection of each axis in LS/SST model. The mapping vector in the middle of Figure 5 is a simple mechanism to retrieve the corresponding node signature in Sequential Signature Table with given vector index. It is important to note that the axis forms a partition of the document, from the point of view of the current node (self). The ancestor, descendant, following, preceding and self axes partition a document, in which the attribute nodes are ignored, they do not overlap and together they contain all the nodes in the document.

Definition 4. XPath axis in LS/SST approach.

1. **Ancestor axis:** The ancestor axis contains the ancestors of the context node. The ancestors of the context node consist of the parent of context node and the parent's parent and so on. Thus, the ancestor axis will always include the root node, unless the context node is the root node. The ancestor nodes can be easily retrieved from logical structure as shown in Algorithm 1. First we assume the index of self node is i, then the ancestor nodes of self node is represented as Ancestor(i).

Algorithm 1. Get ancestor axis of a node

```
function getAncestorAxis (i)
/* INPUT: self node index */
/* OUTPUT: A set of nodes */
{
```

```
Vector result = new Vector();
Node n = mappingVector.elementAt(i);
If (n is root)
    return result; /* root do not have ancestor */
do {
    n = n.getParentNode();
    result.addItem(n);
    } while (n.getParentNode() is true);
return result;
}
```

2. **Preceding sibling axis:** The preceding-sibling axis contains all the preceding siblings of the context node. If the context node is an attribute node, the preceding-sibling axis is empty.

Algorithm 2. Get preceding sibling axis of a node

```
function getPrecedingSiblingAxis (i)
/* INPUT: self node index */
/* OUTPUT: A set of nodes */
{
Vector result = new Vector();
Node n = mappingVector.elementAt(i);
If (n.getParentNode() is not true)
    return result;    /* do not have parent node */
foreach child node in n.getParentNode() {
    if( child node index < i)
      result.addItem( child node);
    }
return result;
}
```

3. Following sibling axis: The following-sibling axis contains all the following siblings of the context node. If the context node is an attribute node, the following-sibling axis is empty.

Algorithm 3. Get follow sibling axis of a node

```
function getFollowingSiblingAxis (i)
/* INPUT: self node index */
/* OUTPUT: A set of nodes */
{
Vector result = new Vector();
Node n = mappingVector.elementAt(i);
If (n.getParentNode() is not true)
    return result;       /* do not have parent node */
foreach child node in n.getParentNode() {
    if( child node index > i)
      result.addItem( child node);
    }
return result;
}
```

4. **Ancestor-or-self axis:** The ancestor-or-self axis contains the context node and the ancestors of the context node. Thus, the ancestor axis will always include the root node.

$$Ancestor-or-self\ \ Axis(i) = Ancestor(i) \times node(i)$$

5. **Descendant axis:** The descendant axis contains the descendants of the context node. A descendant is a child or a child of a child and so on. Thus the descendant axis never contains attribute nodes. We assume the number of descendants of node i (σ(i)) is m.

$$Descendant\ \ Axis(i) = \sum_{k=i+1}^{i+m} node(k) \cap node(k) \neq attribute$$

6. **Descendant-or-self axis:** The descendant-or-self axis contains the context node and the descendants of the context node.

$$Descendant-or-self\ \ Axis(i) = (\sum_{k=i+1}^{i+m} node(k) \cap node(k) \neq attribute) \times node(i)$$

7. **Preceding axis:** The preceding axis contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and excluding attribute nodes.

$$\Pr eceding\ \ Axis(i) = \sum_{K=0}^{i-1} (node(k) \cap node(k) \neq attribute) - Ancestor(k)$$

8. **Following axis:** The following axis contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes. Let the number of nodes is n.

$$Following\ \ Axis(i) = \sum_{K=i+m+1}^{n} (node(k) \cap node(k) \neq attribute)$$

4. OPERATION EXAMPLES

The *path expression (PE)* is a string representation for locating the nodes. Let a *PE* have *s* substrings and separates by "/" in a path expression. We define the first *PE* as PE_1. Then, path expression is represented by "$PE = PE_1 /PE_2/.../ PE_s$". In a case of simple path expression, say "menu/food", PE_1 = *"menu"*, PE_2 = *"food"* and $s = 2$. With PE_1 and PE_2 we can get the hash value from the given hash function *H*, $H_{"menu"}$ = *"0000 0100 1000"* and $H_{"food"}$ = *"0000 0010 0010"* respectively. Then, signature check in level signature vector indicates whether this pattern can be found in this XML document. In this case, since $H_{"menu"} = H_{"menu"} \cap LS(2)$ we know there is a high possibility that PE_1 exists in level 2. And also $H_{"food"} = H_{"food"} \cap LS(3)$, so it is very likely that PE_2 exists in level 3.

The path expression is a very flexible representation, though it can be in a complex form. Wild card may be used to extract more information, for example "menu/*/category". By using the LS approach, we first locate the possible level which has "menu" node (PE_1). As $H_{"menu"} = H_{"menu"} \cap LS(2)$, level 2 will be our entry point. Then, from SST we can easily find out all the child nodes of “menu” in level 3 (PE_2 = “*”), which will be our candidate nodes for PE_3. In this case, they are node index 2 (“food”), index 13 (“food”) and index 23 (“drink”). Since $H_{"category"} = H_{"category"} \cap NS(2)$, $H_{"category"} \neq H_{"category"} \cap NS(13)$, $H_{"category"} = H_{"category"} \cap NS(23)$, the node of index 13 can be ruled out as it does not contain “category”. In order to get the result of “menu/*/category”, we need to visit further down to index 2 and index 23.

LS is used for providing an early notification to avoid unnecessary traversal of a document which is particularly beneficial when a large quantity of XML documents exist. It is likely to be the case in document-centric situation. Furthermore, handling a huge file with SST provides the mechanism to reduce the number of subtrees which need to be visited. It happens when we are dealing with data-centric XML data. In addition, sometimes large numbers of documents and huge files occur at the same time. The efficiency of path query navigation can be greatly improved by adapting LS and SST together.

5. EXPERIMENTS

Extensive experimental studies have been conducted to exam both LS and SST approaches. We have implemented our approaches in Java based on J2EE framework, and carried out a series of performance experiments. We first investigate the size of the index scheme on different data sets, and then go on to a performance study over test queries. For comparison purposes, we also adapt the Jaxen XPath Engine (B.McWhirter & .Strachnan, 2004) in our system to perform query execution on the same data sets.

The experiments were conducted on a PC with Intel Pentium III processor with 866 MHz and 512 MB main memory running Windows XP. The algorithms mentioned in the paper were implemented in Sun Java 2 SDK, Standard Edition 1.4.2. We utilized IBM’s XML4J 3.2.1 (XML parser for Java) for parsing XML documents. The data sets were stored on the local disk. We ran our experiments using three different sets of XML data and queries. The characteristics of the different data sets used in the experiment can be found in next section.

5.1 The Size of LS and SST

Table 2 displays the details with regard to the size of LS/SST on top of the DOM model for each dataset. The comparison of original XML size and total size used in LS/SST is presented in Figure 6. As mentioned earlier, hash values and level signatures are handled together in the main memory. SST, however, is stored separately for each corresponding XML document. Clearly, the size of SST, which is closely correlated with the size of each XML file, occupies the majority of the space in LS/SST. On the contrary, the storage requirement of hash values and level signatures is fairly low. The size of hash values is associated with the number of distinct nodes in a dataset, and the size of level signatures is highly related to the number of files and the depth of each file.

For XMark 0.1 dataset (about 11,875 KB), LS/SST requires 1,108.83 KB of storage. Similarly, for a 31,847 KB dataset like Shakes + XMark 0.2, it generates 3,422 KB data of LS/SST. We noted that the storage requirement of the LS/SST technique is about 10% of the dataset size in general. It is important to realize that LS/SST is in a rather primitive structure just to gain an idea of its size, therefore some compression skills can be applied to minimize the storage requirement of LS/SST.

Table 2. Details of Datasets

Name	Size (KB)	Nodes	Hash Values + LS size (KB)	SST size (KB)
Shakes	7,920	179,763	3.36	1,074
XMark 0.1	11,875	167,911	2.83	1,106
XMark 0.2	23,926	336,333	4.23	2,216
Shakes + XMark 0.2	31,847	516,096	7.4	3,422

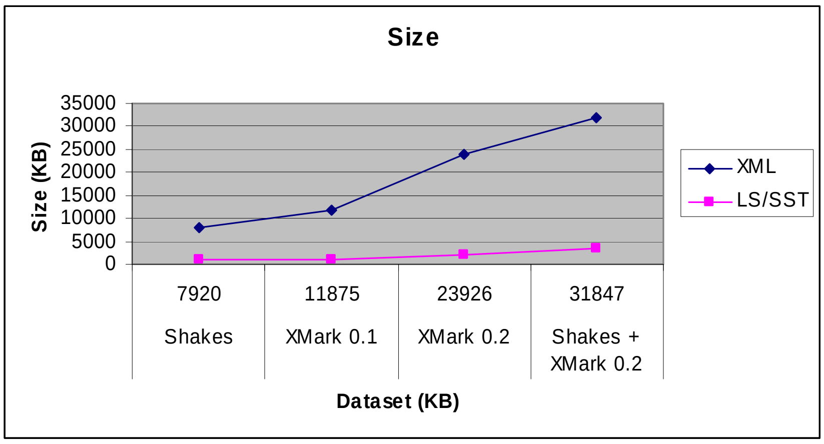

Figure 6. The Size of LS/SST

5.2 Experimental Results

The test queries in Table 3 are highly representative for costly queries as they contain branching patterns. We ran 5 different queries on Shakes dataset and three queries on XMark 0.1 and XMark 0.2. Further, eight queries were performed against Shakes + XMark 0.2 dataset. Our results show that the LS/SST approach outperforms the tree-traversal based approach implemented by Jaxen.

Table 3. Test Queries

Query	Path Query	Shakes	XMark 0.1	XMark 0.2	Shakes + XMark 0.2
Q1	//PLAYSUBT	√			√
Q2	//TITLE	√			√
Q3	//SCENE/TITLE	√			√
Q4	//PERSONA	√			√
Q5	//NON-EXIST	√			√
Q6	/site/regions		√	√	√
Q7	//item/name		√	√	√
Q8	//location		√	√	√

5.2.1 Shakes Play Dataset

Q1 to Q4 demonstrate that selections on the descendant axis by using the "//" symbol require a full tree traversal which requires a longer execution time. SST in this case provides a better mechanism of early notification to avoid unnecessary tree visiting. It is particularly notable when querying Q5 that although the query string "NON-EXIST" does not exist in Shakes Play, the Jaxen approach still needs a full traversal in order to look into every element to make sure of the presence of a string. Due to the similarity of the test queries, the query performances are very much alike. The query execution times for the Shakes Play dataset are shown in Figure 7. It is important to note that both Jaxen and LS/SST are running on top of the DOM model, therefore XML parsing for each file accounts for the majority of time.

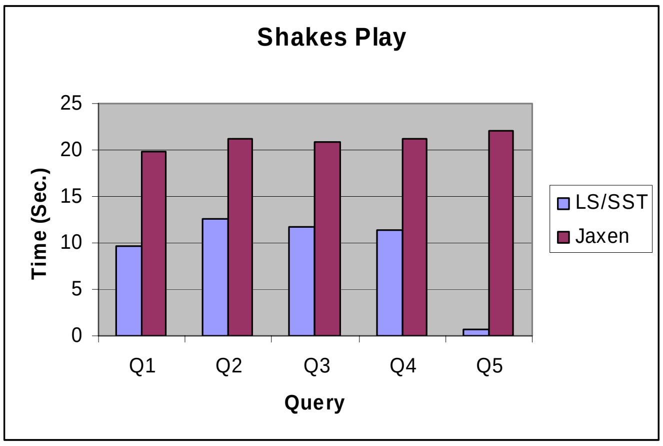

Figure 7. Query Execution Time on Shakes Play Dataset

Table 4. Average Number of Visits in SST

Query	Number of Visits (Avg.)	Nodes in Shakes (Avg.)
Q1	13	
Q2	1,037	
Q3	39	4,858
Q4	45	
Q5	0	

When an unqualified subtree has been found SST can skip this subtree by using the skip function provided in Section 3.3. In the experiment, there are in total 37 XML documents in Shakes Play. For answering Q1, SST visits 13 nodes out of 6638 in "HAMLET.XML" and an average of 13 node visits for each document in Shakes Play. It is quite surprising that Q2 visits 1,037 nodes out of 4,858 nodes in SST. When we looked into the Shakes Play dataset we found that "TITLE" nodes are actually spreading in many different subtrees, such as "/PLAY/TITLE", "/PLAY/PERSONAE/TITLE", "/PLAY/ACT/TITLE", "/PLAY/SCENE/TITLE" etc. That is the reason why the number of visited nodes is significantly greater than others. The average number of visits for each test query is shown in Table 4.

5.2.2 XMark Dataset

We further investigated our approach in querying the XMark datasets. Figure 8 shows the performance results in XMark at scaling factor 0.1 and 0.2, respectively. In Q6 the path is much simpler than other branching and wild cards queries, being straight from root to "site" and then "regions", and therefore LS/SST performs similarly to Jaxen on both XMark 0.1 and XMark 0.2 datasets. Since the size of XMark 0.2 (22.8 MB) is about twice the size of XMark 0.1 (11.3 MB), as expected the execution time is growing linearly with increasing data size.

For a simple path query like Q6, SST visits 2 nodes out of 13,407 in the first file of XMark 0.1 (xmark00000.xml). On average SST only retrieves 2 nodes out of 11,194 in the whole XMark 0.1 dataset. The average number of visits for each query can be found in **Error! Reference source not found.**. Since the number of nodes in each file of XMark dataset varies, we show the nodes in the dataset on average in Table 5. For the branching queries like Q7 and Q8, SST still benefits from skipping unnecessary subtree visits.

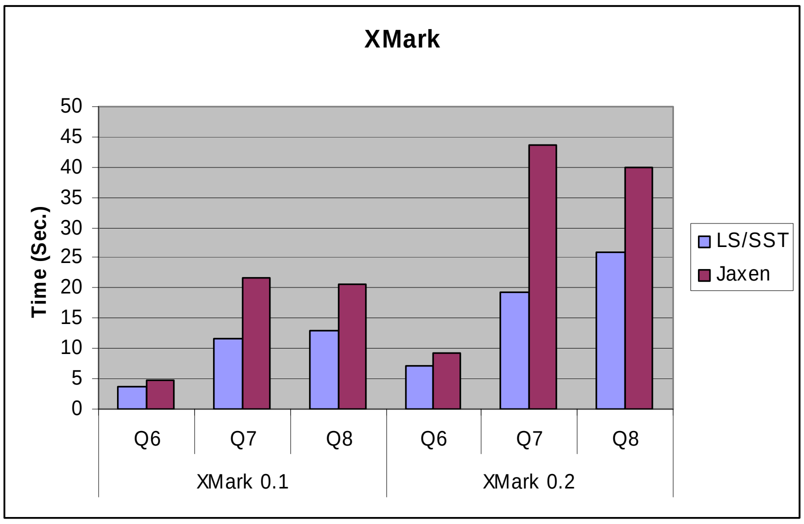

Figure 8. Query Execution Time on XMark Dataset

Table 5. Average Number of Visits in SST

Query	Number of Visits (Avg.)	Nodes in Dataset (Avg.)
Q6	2	11,194 (XMark 0.1)
Q7	3,010	
Q8	4,513	
Q6	2	11,301 (XMark 0.2)
Q7	3,044	
Q8	4,552	

5.2.3 Shakes + XMark 0.2 Dataset

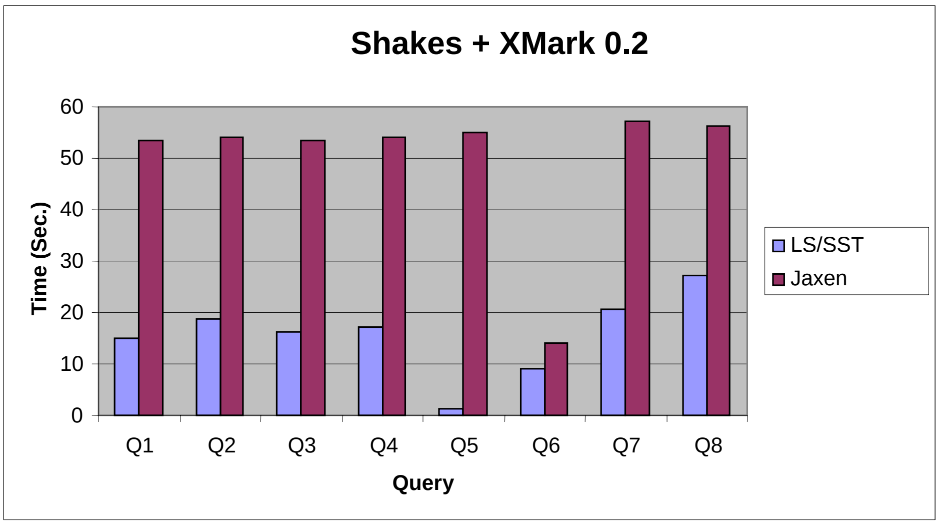

Figure 9. Query Execution Time on Shakes + XMark 0.2

In the experiment we also combined the Shakes Play and XMark 0.2 data sets to create a more complicated data source. The reason we are doing this is that it can represent a document-centric scenario where the documents largely exist, and the idea of level signature can be more useful in reducing access to the documents. The query execution times on combined data sets can be found in Figure 9.

While the execution time of Jaxen implementation increases due to the growth of the data source, our approach excludes a certain number of documents by performing a level signature check before actually accessing files. LS/SST retrieves 27 XML documents out of 64 for answering Q7, as Jaxen needs to access 64 documents. This is probably the best result that could be expected, because the LS approach rules out all 37 Shakes Play data sets without any false drop issue. However, we found the false drops increased dramatically when querying Q2 against XMark 0.2. There are 27 XML documents in XMark 0.2, totalling 22.8 Mb of data. The LS approach only excludes 18 out of 27 XMark 0.2 documents, which means the other 9 documents are incurring false drops. However this is being fixed during the second stage of the SST check. The reason why LS did not function as we expected is that XMark documents consist of a more complicated structure which has up to 13 levels and 13 thousand nodes for each file. We use 32 bits signature in our experiment which may not be enough to represent different nodes. The problem can be improved by increasing signature length or refining the hash function to spread signed bit in a more reasonable way.

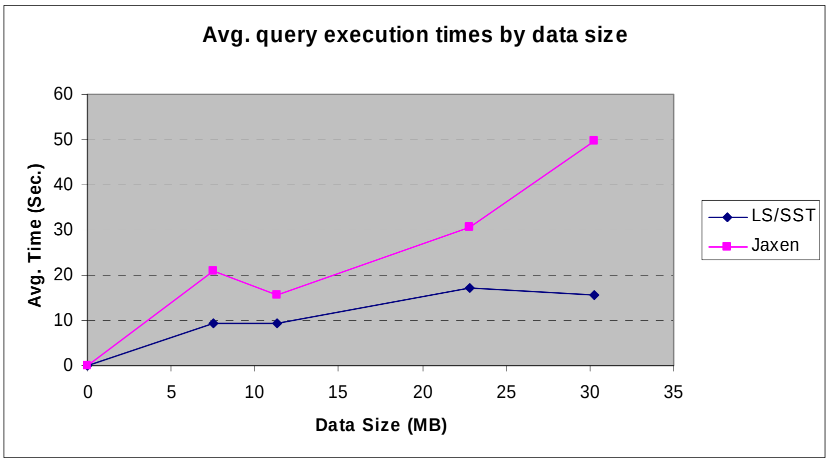

Figure 10. Average Execution Time

Figure 10 shows the average query processing time against data size for LS/SST and Jaxen respectively, where the x-axis shows the data size for Shakes Play (7.5 MB), XMark 0.1 (11.3 MB), XMark 0.2 (22.8 MB) and Shakes+XMark 0.2 (30.3 MB) as listed in Table 17. The execution time of Jaxen drops significantly in 11.3 MB data sets due to a simple path query involved in Q6 which reduces average processing time greatly. As for queries Q1 to Q5 against 7.5 MB data sets, all involved branching queries are much more time consuming than Q6. LS/SST with 30.3 MB dataset has a shorter average execution time than 22.8 MB dataset, since level signature rule out the unqualified files in the first place and the effect of fast processing in Q5. The reduced number of files greatly decreases parsing time as compared with the Jaxen approach.

6. CONCLUSION

Query languages for XML are typically based on regular path expressions that travel the logical XML structure. The evaluation of such flexible expressions is generally expensive. Pattern matching and branching queries, for example, depend heavily on recursive traveling in XML logical structures. The evaluation of such path expressions is a crucial issue to improve the efficiency of XML query evaluation.

In this paper, we propose a novel approach, namely LS/SST, to enhance the ability of path navigation with low storage and construction costs. This approach can be easily implemented into J2EE framework as a service based query execution enhancer. To begin with, each node in XML logical structure has been assigned a hash value, and then the hash value is aggregated to its ancestor nodes to form superimposed signatures in SST. Simply put, the concept is to increase the visibility of a subtree to the parent node. Therefore, the presence of a node can be determined far earlier without visiting the whole subtree. With this knowledge, unnecessary tree visiting can be minimized to a certain level. For LS, the level signature is derived from superimposing the hash values of the nodes in the same level. The level signature checking can identify the existence of a give path in the document level. In summary, LS and SST serve different purposes, the first can rule out XML documents which do not contain a desired path, while the latter can significantly reduce unnecessary traversal in a document.

In this paper, we have described the prototype for our ongoing research. Our experimental results confirm that LS/SST enhances XML query execution performance. This is particularly notable if the query is complex or branchy. We are currently working on developing a complete query processing engine based on this method. We find the idea of using level signature to be quite beneficial due to the fact that XML Query languages, XQuery for example, use "axis" heavily in evaluating queries. And if the DTD information of XML exists, it can give great help for producing signature. In the future we plan to concentrate on query rewriting and optimizing techniques to further improve query execution.

REFERENCES

B.McWhirter & .Strachnan. 2004. *Jaxen: Universal Java XPath Engine*.
Available at: http://jaxen.sourceforge.net/

Chris Faloutsos. 1985. *Signature files: design and performance comparison of some signature extraction methods*. *Proceedings of the 1985 ACM SIGMOD international conference on Management of data* , pp.63-82.

Daniela Florescu & Donald Kossman. September 1999. *Storing and Querying XML Data using an RDBMS*. *IEEE Data Engineering Bulletin* 22[3], pp.27-34.

Haifeng Jiang, Hongjun Lu, Wei Wang, & Jeffrey Xu Yu. 2002. *Path Materialization Revisited: An Efficient Storage Model for XML Data*. *Thirteenth Australasian Database Conference* 24[2], pp.85-94.

Jason McHugh & Jennifer Widom. 1999. *Query Optimization for XML*. *Proceedings of the 25th International Conference on Very Large Data Bases* , pp.315-326.

Jayavel Shanmugasundaram, Eugene Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Efstratios Viglas, Jeffrey Naughton, & Igor Tatarinov. September 2001. *A General Technique for Querying XML Documents using a Relational Database System*. *ACM SIGMOD Record* 30[3], pp.20-26.

Leigh Dodds. October 2001. *XML and Databases? Follow Your Nose*.
Available at: http://www.xml.com/pub/a/2001/10/24/follow-yr-nose.html

Man-Kwan Shan & uh-Yin Lee. 1998. *Placement of Partitioned Signature File and Its Performance Analysis*. *Information Science: An International Journal* 104[3-4], pp.321-344.

Ronald Bourret. July 2004. *XML and Databases*.
Available at: http://www.rpbourret.com/xml/XMLAndDatabases.htm

Sangwon Park & Hyoung-Joo Kim. March 2002. *SigDAQ: an enhanced XML query optimization technique*. *Journal of Systems and Software* 61[2], pp.91-103.

Serge Abiteboul, Roy Goldman, Jason McHugh, Vasilis Vassalos, & Yue Zhuge. 1997. *Views for Semistructured Data*.

Sun Microsystems. July 2003. *White Paper Java Business Integration - A new architecture standard for business integration*.
Available at:
http://developers.sun.com/techtopics/webservices/reference/whitepapers/jbiwhitepaper.pdf

Tim Bray, Jean Paoli, C.M.Sperberg-McQueen, Eve Maler, & François Yergeau. February 2004. *Extensible markup language (XML) 1.0 third edition*.
Available at: http://www.w3.org/TR/REC-xml/

W3C. July 2004. *Document Object Model (DOM) Technical Reports*.
Available at: http://www.w3.org/DOM/DOMTR

Yangjun Chen. May 2002. *Signature files and signature trees*. *Information Processing Letters* 82[4], pp.213-221.