IADIS International Journal on Computer Science and Information Systems
Vol. 1, No. 1, pp. 15-29
ISSN: 1646-3692

EXTENDED TREE-PATTERN CLUSTERING TECHNIQUES FOR MASSIVE XML STORAGES

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

The Extensible Mark-up Language (XML) is an emerging standard for describing data on the Web. As the widespread activities of the Internet and the Web results in vast amounts of data to be generated everyday, the manipulation of such semi-structured textual data is however becoming an important issue in XML storage research. The unique feature of semi-structured data is generally suitable for storage in a tree-like data form. Locating data in this form is based on tree-pattern matching techniques. As a result, effectively evaluating path expression is the key to provide efficient access to such tree-like data storage. In this paper, we apply two novel signature based access methods, which can significantly extend the scope of tree-pattern cluster in order to navigate massive XML databases. We present the process of producing the signatures in details, and further provide the algorithms to demonstrate how they would work. We also show the advantages of using extended tree-pattern clustering techniques in handling large amounts of XML documents.

KEYWORDS

XML, Tree Pattern, Path Expression.

1. INTRODUCTION

XML is an emerging standard for representing data (Tim Bray et al., 2004). The extensible nature of XML provides flexibility in describing variable structure of data. It is becoming a standard data format for data exchange and data integration. XML is also known as an easy-to-write and easy-to-parse language which offers a way to exchange data between varieties of applications on the Internet (Ronald Bourret, 2004).

From the enterprise aspect, 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, resulting in data heterogeneity. Relational database management systems benefit from the universal acceptance of Structured Query Language (SQL) as the primary means of obtaining desired data whilst document and email repositories are generally

accessed using text search engines with varying interfaces and capabilities. In view of the fact that these systems were not designed with interoperability in mind, each must generally be accessed using source specific applications or application programming interfaces (APIs).

XML data is generally defined in a tree or graph based, self-describing object instance model (Serge Abiteboul et al., 1997). Existing approaches for storing semi-structured data fall into two broad categories: Flat streams and Meta-modelling (Carl-Christian Kanne & Guido Moerkotte, 2000). In the former, XML data is stored as serialized byte streams, for example, as XML files in a file system. This method provides fast access when retrieving the entire document or large parts of the document, since by a single index lookup or positioning the disk head once, the entire document or fragment can be retrieved. However, problems can occur when retrieving data in any other form. In the latter, XML data is stored in conventional databases which use the relational database as its data model. Since XML provides the flexibility for storing data with differing structures, the semi-structured data is incompatible with the flat structure of conventional database tables. Therefore, using the relational database for handling XML usually introduces performance overheads particularly in document reconstruction. For this reason, our research is focused on document centric situation where a large number of documents exist that might be small or extremely large in size. Further, for querying XML documents a number of query languages has been proposed, such as Lorel (Jason McHugh et al., 1997), XML-QL (Alin Deutsch et al., 1998), Quilt (Jonathan Robie et al., 2000) and XQuery (Scott Boag et al., 2003). They have one characteristic in common, that they support regular path expressions to navigate through the logical, hierarchical structure of an XML document. Evaluation of the regular path expressions is normally costly, as it 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). Therefore, the efficiency of evaluating path query plays an important role in XML query processing. 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 improve query execution performance significantly.

The rest of the paper is organized as follows. Section 2 gives a brief description of related work. Section 3 discusses the observations that led us to the introduction of the new methods. The proposed methods are described in detail along with the algorithms 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. RELATED WORK

A considerable amount of work has been undertaken on indexing for semi-structured data (Brian Cooper et al., 2001;Chin-Wan Chung et al., 2002;Roy Goldman & Jennifer Widom, 1997;Tova Milo & Dan Suciu, 1999). Many techniques have been developed to support label path expressions and have concentrated on indexing simple path expressions. Goldman and Widom proposed a path index called strong DataGuide (Roy Goldman & Jennifer Widom, 1997). The strong DataGuide is restricted to a simple label path and is not useful in complex path queries with several regular expressions (Tova Milo & Dan Suciu, 1999). The

construction of a DataGuide is like the conversion of a non-deterministic finite automation (NFA) into a deterministic finite automation (DFA). This conversion takes linear time when the source is a tree and exponential time in the worst case when the source is a graph (Chin-Wan Chung et al., 2002). One of the problems in building a DataGuide of a deeply nested graph is that it may end up creating a node for every subset of nodes in the data source. Therefore, the size of data created by a DataGuide could grow exponentially to be larger than the original data.

Milo and Suciu proposed another index family, the 1/2/T-index (Tova Milo & Dan Suciu, 1999). 1-index keeps track of all the absolute paths from root node. It represents the same set of paths as the DataGuide. It can be seen as a non-deterministic version of the strong DataGuide. 1-indexes focus on the queries which search for nodes matching some arbitrary path expression in the database from the root. 2-indexes cover all the relative paths and focus on the queries which search for pairs of nodes matching some arbitrary path expression. T-index is a template-based indexing technique. The idea of T-index is similar to that of access support relations in object-oriented databases. T-index builds indexes not on all paths, but on selected path templates. When the data is very irregular and cyclic, indexing all the paths as with the 1/2-index may become too large and inefficient. Restricting the class of queries supported by the index structure can reduce the index complexity and yield better performance.

The Index Fabric was introduced by Cooper et al (Brian Cooper et al., 2001). It shares an idea similar to the strong DataGuide in keeping all label paths starting from the root element, but with a better extension to replace values with identifiers. Each label path is first encoded with one or more letter. Each label path to each XML element is then encoded as a sequence of encoded labels followed by the data value as a string. For example, the path /Menu/Food/Calories[970] is encoded as MFC970. The method stores the encoded strings in an efficient index such as the Patricia trie. A Patricia trie is a simple form of compressed trie that merges single child nodes with their parent nodes. The Patricia tree embeds an inherently balanced mechanism to guarantee constant access time in path lookups. Path expressions including predicates on values are performed as a string search. It is particularly beneficial when the structure of the data is changing, variable or irregular. However, since the index only records the elements with value (it does not keep information on non-leaf nodes and the order of elements in documents) it has performance overheads in handling partial matching path queries.

Chung et al. proposed an adaptive path index for XML data named APEX (Chin-Wan Chung et al., 2002). While other methods, such as DataGuide and 1/2-index, maintain all paths from the root node, APEX only adopts frequently used paths to improve the query performance. APEX is constructed by adopting a data mining algorithm, a sequential pattern mining technique, to summarize paths that appear frequently in the query workload. An APEX index consists of two structures, a graph structure representing the structural summary of the data, and a hash tree structure that associates required paths to nodes of the graph structure. For efficiency, each node of the hash tree is implemented as a hash table on the labels. Terminal nodes refer to a graph structure node. The hash tree is used both to find nodes of the structure graph for a given label path and for incremental updates.

To sum up, we need to pay more attention to several issues. 1) Size of indexes. 2) Document update. 3) Partial match. 4) Structural query. First of all, the tradeoff between the size of indexes and performance is an important issue in database related fields. Increasing space usage of indexes usually involves overlapping parts which are stored redundantly. The

approaches, such as DataGuide and 1/2-index, will produce a large index size when the data is highly nested. That can result in degradation of query evaluation, since evaluating a query on a large index is just like evaluating the query on the base data (Raghav Kaushik et al., 2002). Also, the construction and re-construction of the index is often costly, particularly if the data needs to update frequently. Secondly, many queries on XML data involve a partial match, such as branching query with the self-or-descendent axis ("//") and wild cards ("*" and "?"), which requires exhaustive navigation of the data source. It can be even more complicated when it comes with structural query which structural relationships specified in the query, namely, parent-child and ancestor-descendant relationships. In such circumstances the path exploration can grow exponentially. Therefore, we are attempting to tackle this problem by avoiding unnecessary traversing. Our LS/SST approach exposes the subtree to the parent node, in order to provide an early notification of the existence of a node in the subtree.

3. APPLYING SIGNATURES TO TREE PATTERNS

Table 1. Example hash values derived from hash function *H*

Hash function	Hash value
H(menu)	0000 0100 1000
H(food)	0000 0010 0010
H(drink)	0000 1000 0010
H(category)	0001 0000 1000
H(name)	0000 1010 0000
H(price)	0010 0000 0010
H(description)	0000 1001 0000
H(calories)	0001 0001 0000
H(chef)	0010 1000 0000
H(id)	0000 0000 0011
H(first)	1000 0000 0001
H(family)	1100 0000 0000
H(email)	0000 0000 1010
H(middle)	1000 0001 0000
H(phone)	0100 0000 0100

During the last few years a number of index approaches have been intensively studied in order to provide better manipulation of data. 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. In this paper, we apply the signature approach to build tree patterns in order to provide new access methods. 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 pattern of the XML DOM tree. The details will be discussed in Section 4.

We first 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. 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. Table 1 shows the example hash values. Algorithm 1 gives the idea for producing the signature of a node.

Algorithm 1. Produce the signature of a node

```
1.  function getNodeSignature (node)
2.  /* INPUT: node */
3.  /* OUTPUT: node signature NS */
4.  {
5.  NS = 0;
6.  foreach DescendantNode in node {
                    /* get all nodes in the subtree */
7.              if  (DescendantNode is an Element or
                Attribute node ) {
8.              NS = NS U Hash(DescendantNode.Name);
                      /* bitwise OR operation*/
9.               }
10.           }
11. return NS;
12. }
```

4. PROPOSED METHODS

In this paper we propose a generic access method by clustering signature tree-patterns in two aspects, document level and subtree level, respectively. Firstly, we assume an XML document is represented as a DOM tree, an example of which can be found in Figure 1, 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 creating the signature.

4.1 Document Level Signature (DLS) Clustering

The *Document Level Signature (DLS)* clustering is the OR-ed signature of the nodes that reside in the same level of the DOM tree. Moreover, the *Document Level Signature Path (DLSP)* is the path of document level signature arranged according to the level of the DOM tree. Figure 1 shows an example of DLS clustering processes for XML document *D1* and *D2*.

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

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

Definition 2. (Document Level Signature Path): We assume the document level signature path of an XML document *D*, denoted by $DLSP_D$, where the height of document *D* is *h*.

$$DLSP_D = DLS(1) / DLS(2) / \ldots / DLS(h)$$

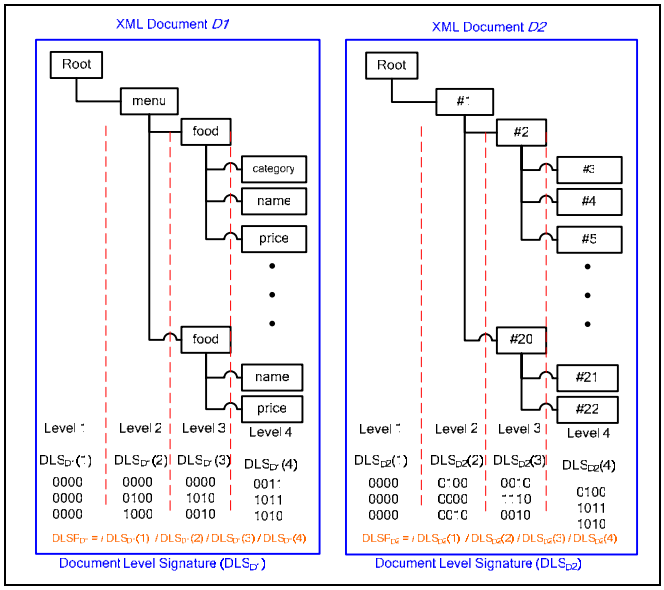

Figure 1. Document Level Signature (DLS) clustering for XML Document D1 and D2

4.2 Subtree Level Signature (SLS) Clustering

The *Subtree Level Signature (SLS)* clustering 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 *Subtree Level Signature* 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 2. Figure 2 shows an example of SLS clustering procedure for XML document D1.

SLS(Level, Index, NodeSignat ure)

Definition 3. (Subtree Level Signature): The subtree level signature of *i*th node of document *D*, denoted by $SLS_D\ (i)$ 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.)

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

The SLS is in a sequential form of tree structure, and as mentioned earlier it is also a depth first traversal. Because of that there is an interesting feature in the SLS formation, which physically clusters the subtree together on a sequential basis. This property offers an efficient way to identify the subtree clustering. We then show this concept in Definition 4 below.

Definition 4. (Subtree of a node): The *i*th node in SLS denoted as *node(i),* where the 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), …, node(i + m)

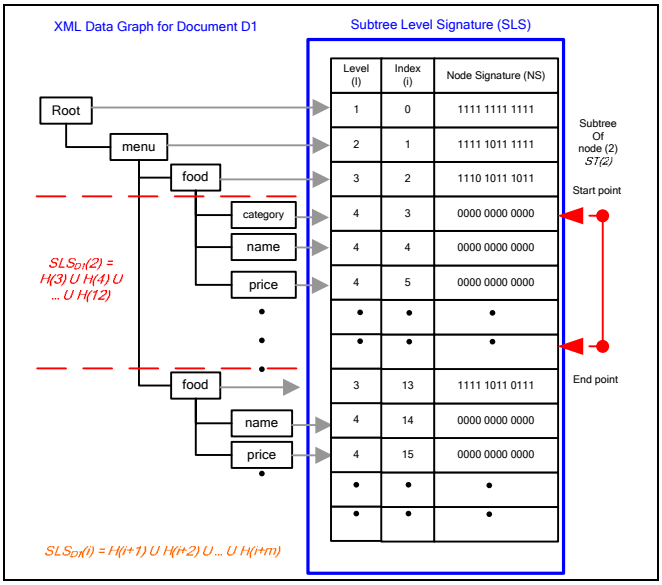

Figure 2. Subtree Level Signature (SLS) clustering for XML Document D1

4.3 The Algorithms for Creating DLS and SLS

As mentioned earlier in this section, the index in SLS 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 SLS clustering, the idea of using index plays an important role in keeping tree structure and node signatures in line. From definition 3, the node signature (NS) of SLS is derived from ORing all the hash values of the nodes in the subtree (all descendants). Algorithm 1 in Section 3 depicts the producing process of node signature. Algorithm 2 is an example to create an SLS instance in an array implementation.

Algorithm 2. makeSLS (rootNode)

```
1.  function makeSLS (rootNode)
2.  /* INPUT: root node */
3.  /* OUTPUT: SLS in an array */
4.  {
5.  m = NumberOfNodes( rootNode)
6.  for( i = 0; i < m; i++) {
7.          SLS[i] [0] = node(i).getLevel;
8.          SLS[i] [1] = i;
9.          SLS[i] [2] = getNodeSignature( node(i) );
10.             }
11. return SLS;
12. }
```

As the level is clearly defined in the *Subtree Level Signature (SLS)* clustering, it provides a better mechanism to retrieve all the nodes in the same level without visiting the entire tree structure in order to calculate *Document Level Signature (DLS)*. Here we give an example of creating DLS in a vector as shown in Algorithm 3.

Algorithm 3. makeDLS (SLS)

```
1.  function makeDLS (SLS)
2.  /* INPUT: subtree level signature */
3.  /* OUTPUT: document level signature vector */
4.  {
5.  Vector DLS = new Vector( );
6.  foreach i in SLS {
7.              l = SLS[i][0];
8.              if  ( DLS.elementAt(l) Exist)
9.   DLS.elementAt(l) =   DLS.elementAt(l)  U getNodeSignature( node(i) );
10.             /* bitwise OR operation*/
11.         else DLS.insertElementAt( getNodeSignature( node(i) ), l);
12.         }
13. return DLS;
14. }
```

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 Definition 4 equation. Then we need to know the position at which that subtree ends. Level in SLS 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. An example of the subtree of *node(2)* can be found on the right hand side of Figure 2, in this case the subtree is from *node(3)* to *node(12).* Algorithm 4 shows this concept that applies to the calculation of the number of descendants.

Algorithm 4. getNumberOfDescendant (node)

```
1.  function getNumberOfDescendant (node)
2.  /* INPUT: a given node */
3.  /* OUTPUT: NumberOfDescendant */
4.  {
5.  nodeIndex = node.getIndex();
6.  l = SLS[nodeIndex][0];
7.  i = nodeIndex + 1;
8.  /* nodeIndex + 1 is the start position of subtree */
9.  while ( SLS[i][0] > l) {
10.             i++;
11.             }
12. return (i – nodeIndex);
13. /* i is the index of the end position of subtree */
14. }
```

5. EXPERIMENTS

Extensive experimental studies have been conducted to exam our DLS and SLS tree-pattern clustering approaches. We have implemented our approaches in Java, and carried out a series of performance experiments in order to observe the effectiveness of the approaches. We first investigate the construction time of the DLS and SLS on different data sets, and then go on to a performance study over queries. The different characteristics of each query are also discussed and reported in detail.

5.1 Experiment Setup

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. They are summarized below. The characteristics of the different data sets used in the experiment can be found in Table 2.

- Shakes: The first data set is the Jon Bosak collection of the plays of Shakespeare (Jon Bosak, 2004). The Shakes Play XML data shows a minor irregularity in the structure. Since the Play does not involve ID and IDREF, it is a pure tree structured XML data. There are 37 documents, totaling 7.5 Mb of data.
- XMark: The second data set is from the XML Benchmark project (A.R.Schmidt et al., 2001). The XMark data models an auction website, and the document consists of sub structures such as item (objects for sale), person (buyers and sellers), category, open_auction, and closed_auction, etc. The tag names are highly self-explanatory and the tag itemref, for example, is an IDREF value pointing to item nodes. XMark dataset has more complicated structure compared to Shakes. In our experiments, we use two XMark data sets generated by xmlgen (A.R.Schmidt et al., 2001) with a scaling factor 0.1 , 0.2 and 0.4, respectively.
- Shakes + XMark 0.2: The third data set combines Shakes Play and XMark data with scaling factor 0.2 to represent a more complicated document-centric situation. The data is highly irregular and large numbers of XML files are involved in query answering.

Table 2. Details of the data sets

Name	Size (MB)	Nodes
Shakes	7.5	179,763
XMark 0.1	11.3	167,911
XMark 0.2	22.8	336,333
Shakes + XMark 0.2	30.3	516,096
XMark 0.4	46.4	684,442

Table 3. Test queries

Query	Path Query	Shakes	XMark 0.1, 0.2, 0.4
Q1	/PLAY	√	
Q2	/PLAY/ACT/SCENE/SPEECH	√	
Q3	/PLAY[PLAYSUBT = 'HAMLET']	√	
Q4	//*/TITLE	√	
Q5	//NON-EXIST	√	
Q6	/site		√
Q7	/site/regions/*/item/name		√
Q8	//item[location = 'United States']		√
Q9	//item/name		√

5.2 Performance Results

5.2.1 Building Time

The building time of *Document Level Signature* and *Subtree Level Signature* is basically derived from the sum of the time spent on XML parsing and generation time of hash values, DLS and SLS. We assume that there are n XML files in the given directory. The formula below indicates how the DLS/SLS building time is summed up.

$$T_{total} = \sum_{n=1}^{n} T_{par\sin g}(n) + T_{hash\_value}(n) + T_{SLS}(n) + T_{DLS}(n)$$

We built five datasets separately with DLS/SLS tree-pattern clustering techniques. They are Shakes, XMark 0.1, XMark 0.2, Shakes + XMark 0.2 and XMark 0.4, respectively. The building time for each dataset is summarised in Figure 3. From the experiment we observe that the building time increases with a nearly linear growth of the size of datasets. When the source XML document size is 46.4M bytes, the building time doubles compared with the dataset in 22.8M bytes. Although the size of XMark 0.1 is 1.5 times bigger than the Shakes dataset, the building time difference does not change notably. It is due to the characteristic differences between Shakes and XMark. The Shakes dataset has a simpler structure with more textual data content; while in contrast, the XMark dataset has a more complicated structure with less textual data content. It appears that most of the building time is spent in generating *Document Level Signature* and *Subtree Level Signature* in most cases. To be more precise the generation of the DLS is far cheaper than the SLS. The time spent in creating SLS is highly correlated with the complexity of tree structure. The Shakes dataset contains the majority of text which costs less time when creating DLS and SLS but more time on $T_{parsing}$.

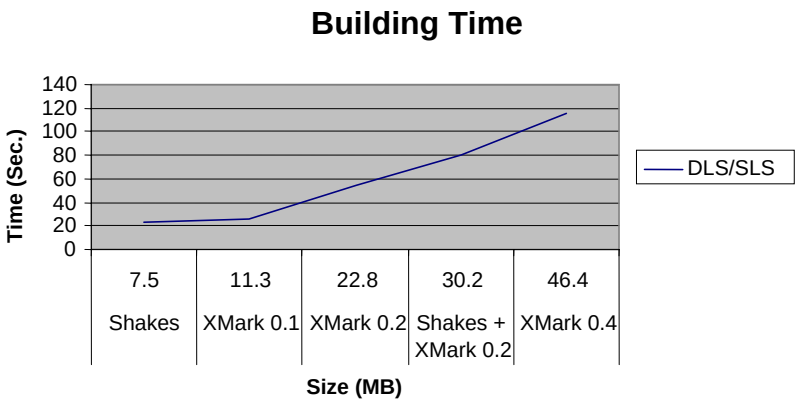

Figure 3. Native VXD Building Time

Note that the building time of the forth dataset, which is Shakes + XMark 0.2, is 80.1 seconds. It is roughly the sum of the individual building times of Shakes (23.7 sec.) and XMark 0.2 (54.6 sec.). That makes sense, because of the fact that $T_{parsing}$(Shakes + XMark 0.2) equals to $T_{parsing}$(Shakes) plus $T_{parsing}$(XMark 0.2). It appears the same way for T_{hash_value}, T_{DLS} and T_{SLS}. The only difference is the switching cost between Shakes and XMark 0.2.

5.2.2 Query Execution Time

The query execution time for Q1 to Q5 using Shakes Play dataset is shown in Figure 4. Query Q1 is a short and simple path query directly from document root which means that there is no need for the query executor to visit the whole document. In Shakes Play dataset each XML file contains exactly one result for the path of *"/PLAY"*. The DLS/SLS technique outperforms the DOM model because the logical structure is handled separately in *Subtree Level Signature* Cluster. Query Q3 functions in a similar way, but in addition contains an exact match of *"HAMLET"*. Q1 and Q3 are the first to provide surprises in the experiment. It turns out that the DLS/SLS technique is not really helpful when the query is particularly simple. For a simple query with a longer path such as query Q2, SLS cuts down the processing time by 24% compared with the DLS approach alone. The DOM Model requires 3.6 times more processing time than SLS technique. Q2 takes more time than other queries due to its larger result set, totalling 30,933 nodes.

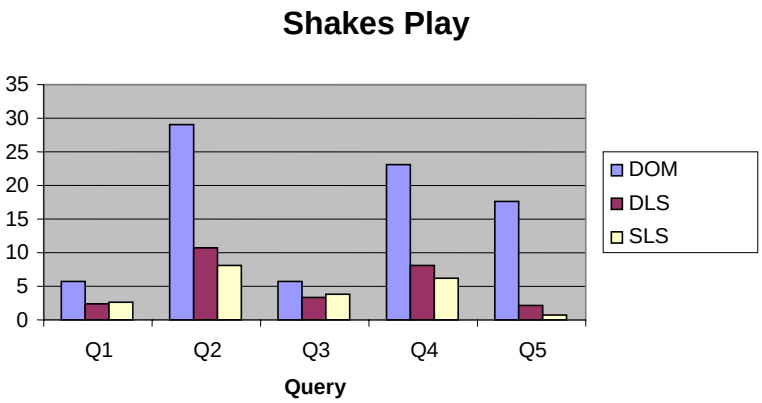

Figure 4. Query Execution Time Using Shakes Play Dataset

Query Q4 is a typical branching query with a wildcard "*". Although the result set of Q4 is 1,031 nodes, which is much smaller than 30,933 nodes in Q2, the processing time of Q4 is still costly in the DOM model. The DOM model suffers from requiring a traversal in depth. Conversely, the DLS/SLS takes some advantage in knowing the existence of a node earlier to avoid unnecessary travelling. SLS provides about a 25% performance boost up in evaluating query Q4. A simple branching query with a descendant axis at the leading place like query Q5, the DOM model requires a full tree structure scan, since it is possible that the given path is located in the very end of tree.

Despite the fact that the "NON-EXIST" string is not in any file of Shakes Play, the DOM model needs 17.6 seconds of processing time in order to get the result. As in SLS technique, the given path query cannot pass the hash value examining (Hash Exam) at the very beginning stage. As a result, SLS significantly outperforms the DOM model (up to 22 times faster).

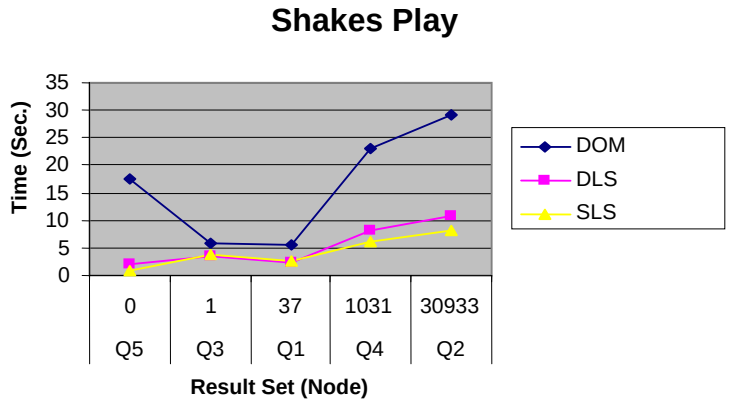

Figure 5. The Result Set of Test Query

In general, the bigger result sets require more time for the query executer to locate, but is not always true considering different types of queries. Picturing a complicated query with one result, it might take much longer than a simple query with considerable amount of results. Figure 5 shows the relationship between the size of result sets and query execution time. For similar queries, such as Q1 to Q3, the query evaluating time remains stable with consideration to the size of result sets. Without considering Q5, which is a rather special case, the execution time grows dramatically when querying with the DOM model with large results. The DLS and SLS are probably having more advantages in navigating paths and locating qualified nodes. The DOM model requires parsing XML document each time, as in DLS and SLS we only need to load corresponding mappings and signature files. The data content can be located only when it is required, for example in query Q3.

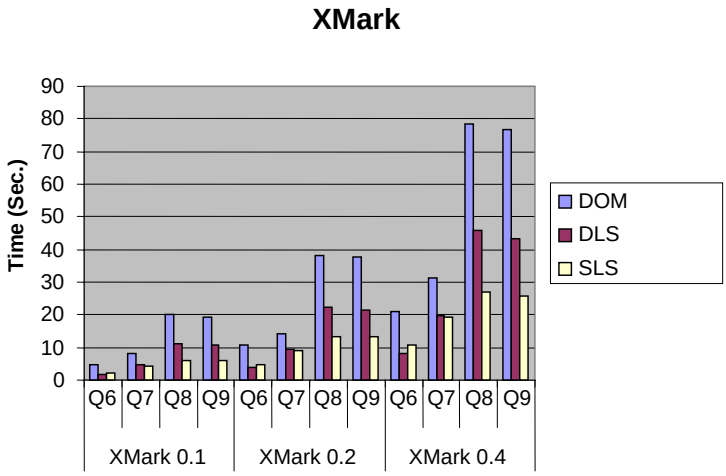

Figure 6. Query Execution Time Using XMark Dataset

Figure 6 represents the results of query Q6 to Q9 based on XMark 01, 0.2 and 0.4, respectively. Query Q6 is a short and simple path query, and the result set is very small (15 nodes). The execution time of Q6 has a linear growth rate when performing a query on the increased dataset.

Query Q7 is a simple path query with a longer path and wildcard, and Q8 is a branching query starting from descendant axis. It is important to note that the results of Q7 and Q9 are identical. However, the execution times of query Q9 are twice those in most cases in comparison to Q7. Evaluating a branching query, such as Q9, is far more costly than a simple path query (Q7), since a complete scan of the document structure may be required. SLS technique outperforms others in most cases, except Q6. Q6 is quite simple, but SLS needs to do extra hash examining and signature testing compared with DLS. Therefore, the performance differences between DLS and SLS are not distinguished on both query Q6 and Q7. That means the query executer performs similarly with or without the aid of DLS and SLS techniques. The performances of Q8 and Q9 are very much alike in all the approaches, due to the similarity of these two queries.

6. SUMMARY AND FUTURE WORK

In this paper, we proposed two signature based approaches, namely DLS and SLS, which are easily adapted to data models for efficiently accessing XML documents. In a data-centric situation, data is stored with a certain structure and probably within one extremely large file, however in a document-centric environment where documents largely exist in relatively complex structures (Leigh Dodds, 2001). We have examined both aspects and provided simple solutions. DLS and SLS serve different purposes, the first can rule out XML documents which do not contain desired information, while the latter can significantly reduce unnecessary traversal in a document. Also they can work complementally, as DLS can first rule out the unqualified documents, and SLS can then be used to avoid unnecessary subtree visits.

In summary, our approaches have several desirable features: 1) it provides a simple mechanism to map into the data model, 2) it reduces unnecessary file traversal, 3) it minimizes redundant subtree visiting, 4) it skips unnecessary signature checking and 5) it has low storage overheads. In this paper, we have described the prototype for our ongoing research. Our experimental results confirm that DLS/SLS provides an efficient way for tree-pattern navigating in XML data. 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 *Document 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 signatures. In the future we plan to concentrate on query rewriting and optimizing techniques to further improve query execution.

REFERENCES

A.R.Schmidt, F.Waas, M.L.Kersten, D.Florescu, I.Manolescu, M.J.Carey, & R.Busse. April 2001. *XMark -- An XML Benchmark Project. Technical Report*. Available at: http://monetdb.cwi.nl/xml/index.html

Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, & Dan Suciu. August 1998. *XML-QL: A Query Language for XML*. Available at: http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/

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

Brian Cooper, Neal Sample, Michael J.Franklin, Gisli R.Hjaltason, & Moshe Shadmon. September 2001. *A Fast Index for Semistructured Data*. *Proceedings of the 27th International Conference on Very Large Data Bases* , pp.341-350.

Carl-Christian Kanne & Guido Moerkotte. March 2000. *Efficient Storage of XML Data*. *Proceedings of the 16th International Conference on Data Engineering* , pp.198.

Chin-Wan Chung, Jun-Ki Min, & Kyuseok Shim. June 2002. *APEX: An Adaptive Path Index for XML Data*. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* , pp.121-132.

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.

Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, & Jennifer Widom. February 1997. *Lore: A Database Management System for Semistructured Data*.

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.

Jon Bosak. 2004. *The Plays of Shakespeare*. Available at: http://metalab.unc.edu/bosak/xml/eg/shaks200.zip

Jonathan Robie, Don Chamberlin, & Daniela Florescu. March 2000. *Quilt: an XML Query Language*. Available at: http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html

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.

Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, & Henry F Korth. June 2002. *Covering Indexes for Branching Path Queries*. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* , pp.133-144.

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

Roy Goldman & Jennifer Widom. August 1997. *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*. *Proceedings of 23rd International Conference on Very Large Data Bases* , pp.436-445.

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

Scott Boag, Don Chamberlin, Mary F.Fernández, Daniela Florescu, Jonathan Robie, & Jérôme Siméon. August 2003. *XQuery 1.0: An XML Query Language W3C Working Draft*.

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/

Tova Milo & Dan Suciu. January 1999. *Index Structures for Path Expressions*. *Proceeding of the 7th International Conference on Database Theory* , pp.277-295.

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.