A New Query Processing Technique for XML Based on Signature[*]

Sangwon Park
School of Computer Science and Engineering
Seoul National University, Seoul, Korea
swpark@oopsla.snu.ac.kr

Hyoung-Joo Kim
School of Computer Science and Engineering
Seoul National University, Seoul, Korea
hjk@oopsla.snu.ac.kr

Abstract

XML is represented as a tree and the query as a regular path expression. The query is evaluated by traversing each node of the tree. Several indexes are proposed for regular path expressions. In some cases these indexes may not cover all possible paths because of storage requirements. In this paper, we propose a signature-based query optimization technique to minimize the number of nodes retrieved from the database when the indexes cannot be used. The signature is a hint attached to each node, and is used to prune unnecessary sub-trees as early as possible when traversing nodes. For this goal, we propose a signature-based DOM(s-DOM) as a storage model and a signature-based query executor(s-NFA). Our experimental results show that the signature method outperforms the original.

1. Introduction

XML is an emerging standard for data representation and exchange on the World-Wide Web. A database system is required for efficient manipulation of XML data, as large quantities of information are represented and processed as XML. However, because the data model of XML is different from those of conventional databases, a new storage method and a query processing model are required. Semi-structured data[1, 5], which has been intensively studied in recent years by the database research community, is very similar to XML data. Therefore, the research results in the area of semi-structured data are now broadly applicable to XML[20]. There are several semi-structured or XML database systems, e.g., Lore[19] and eXcelon[11].

XML is represented as a tree of which each node is stored as an object in the semi-structured database, and queries are evaluated by traversing these nodes. For efficient evaluation of the XML query, it is important to decrease the number of the traversed nodes.

*This work was supported by the Brain Korea 21 Project.

```
SELECT x.company.(address|telephone)
FROM person.*.parent x;
```

The above is an example of an XML query, which is similar to Lorel[2]. This query retrieves person's parents' addresses or telephone numbers. It contains the regular path expressions[2, 6, 9], which are supported by general XML queries such as XML-QL[10] and XQL. Some syntaxes, such as the star(*) in XML queries, enlarge the search space. In this example, almost all nodes under `person` must be visited because of `person.*`. Therefore, regular path indexes have been studied to solve this problem.

The path index[4] is proposed for evaluating path expressions in object-oriented databases. However, all possible paths cannot be covered by this index due to the high storage requirements. New indexing methods for semi-structured data are proposed in [16, 22] to evaluate the regular path expressions more rapidly. The 2-index[22] is for $*.x.P.y$, in which P means a regular expression. However, in the worst case, the number of nodes in the 2-index is the square of the number of nodes in the data graph. For this reason, the T-index[22] is introduced to decrease the size of 2-index by reducing the coverage of regular expressions such as $*.person.x.P.y$.

As a result, some data are outside the boundary of these indexes. The path index and T-index do not cover all possible regular path expressions for the storage requirements. It is also a problem that the index for a semi-structured data is another semi-structured data[16, 22]. When the index is used for query evaluation, the index nodes must be traversed. However, the number of visited index nodes cannot be reduced even though they are index nodes.

An s-DOM and s-NFA are proposed which are based on the signature method[8, 12], to reduce the search space when the index is not used for the regular path expressions. The signature of s-DOM gives a hint as to whether some nodes exist in the sub-tree of a specific node. The s-NFA is used for evaluating the regular path expressions using the signature information. This method can be applied to semi-structured indexes because they are also represented as a graph. To evaluate the regular path expressions many of the nodes in the indexes have to be visited because of blindness of a sub-graph to a node in the index. The signature method removes the blindness, and reduces the number of visiting nodes of the data and index trees.

The size of each node of s-DOM becomes larger than the size of the original because a signature is stored in each node. However, as the size of a signature is several bytes, the performance is not much affected. Because the operation of signatures is a bit-wise operation, there is little overhead for computation of the signature.

The remainder of this paper is organized as follows: Section 2 presents related work, while Section 3 defines the data model and the query language used in this paper. Section 4 presents the s-DOM for nodes that have signatures. The query optimization technique using signatures is given in Section 5 and the experimental results are discussed in Section 6. Finally, conclusions are presented in Section 7.

2. Related Work

Semi-structured data[1, 5] is represented as a graph. The query languages for semi-structured data are influenced by those of object-oriented databases such as OQL[7], XSQL[17]. Both OQL and XSQL use a path expression which enhances the expressive power of the queries. However, these query languages are not adequate for the semi-structured data due to a lack of schema information. Even if schema information is provided, the structure can be changed by its own data.

To solve this problem, regular path expressions are used for semi-structured queries[2, 6, 9]. Indexes of semi-structured data[16, 20, 22] are proposed to execute regular path expressions more rapidly. They combine the index structure and automata of the XML data. The target objects can be retrieved by traversing the appropriate automata graph for the regular path expression.

Theoretical foundations for query processing for semi-structured data are studied in [3, 21]. [3] uses path constraints for optimization of regular path queries. [13] defines a graph schema that has partial information about the graph structure. It reduces the search space by query pruning and query rewriting.

Each node of a tree is stored as an object in eXcelon[11] and PDOM[15]. The original structure of XML documents cannot be changed by storing each node as an object. Object-oriented databases or Lore[2] use this method.

3. Data Model and Query

The DOM[24] is a standard interface of XML data, whose structure is a tree, which is the data model used in this paper. Each node in DOM references its parent, child and sibling nodes, and the sibling nodes are an ordered list. We assume that each node in DOM is stored as an object. When each node is stored as an object in a database, minimizing node visits is the main requirement to optimize the queries.

```
<?xml version="1.0"?>
<!DOCTYPE AddrList>
<AddrList>
  <person name="Robert Johnson">
    <company>
      <address>Heidelberg</address>
      <telephone>123-4567</telephone>
    </company>
    <father>
      <person>
        <name>William Johnson</name>
      </person>
    </father>
  </person>
  <company>
    <name>Samsung</name>
    <address>Suwon</address>
    <telephone>549-0987</telephone>
  </company>
</AddrList>
```

Figure 1. Example of an XML Document

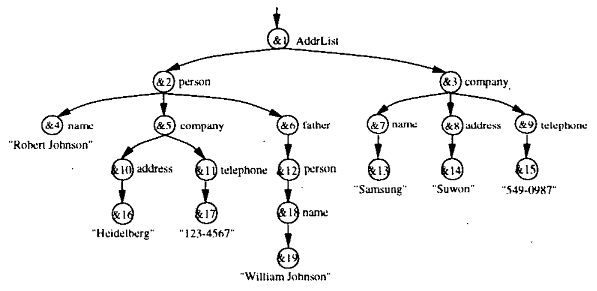


Figure 2. DOM Graph

Figure 1 is an example of an XML document, of which DOM structure is represented as a tree like in Figure 2. Each node is stored as an object and its OID is represented by '&' as depicted in Figure 2. For example, the OID of the root node is &1. There are element node, attribute node and text node in DOM. The element node has a name. The text node is a leaf node and has a value, and the attribute node has a name and a value. For example, object &1 and &2 are examples of element nodes, and object &4 is an attribute node. The leaf node, such as objects &16 and &17, is a text node. Simple definitions useful for describing the mechanism described in this paper are:

father	00000011	person	00100010
name	10001000	company	00001001
address	01000001	telephone	00101000

(a) Hash value of string

&1	11101011	&2	11101011	&3	11101001
&4	00000000	&5	01101001	&6	10101010
&7	00000000	&8	00000000	&9	00000000
&10	00000000	&11	00000000	&12	10001000

(b) Signature of a node in s-DOM

Table 1. Hash values of the Name of each Element and the Signatures of each Node

Definition 3.1 (label path) *A label path of a DOM is a sequence of one or more dot-separated labels, $l_1.l_2...l_n$, such that we can traverse a path of n nodes $(n_1...n_n)$, where node n_i has label l_i, and the type of node is element or attribute.*

Definition 3.2 (regular path expression) *A regular path expression is a path expression that has regular expressions in the label path.*

Queries in this paper are regular path expressions. They allow wildcard operators such as *, + and ?. The scan operator is provided for searching nodes matched to the given regular path expression when processing the query. If each node is stored as an object in unclustered fashion it is highly likely that a page is read from disk to fetch a node. Therefore, the number of fetching nodes must be diminished to reduce the cost of evaluating the queries. The objective of this paper is to reduce the search space of the DOM tree by pruning the data graph to minimize disk operation when evaluating regular path expressions.

XML queries can be executed by traversing each node of the tree. Therefore, to optimize XML queries, minimizing the number of visited nodes is the key issue. In this paper the terms node and object are interchangeable because a node is stored as an object in a database.

4. Storing XML Documents Based on the Signature Method

In this paper we assume that each node of DOM is stored as an object, which is shown in [11, 14, 19, 23]. We additionally add a signature to each node in DOM, and call it s-DOM. The label path contains the names of the element or attribute nodes in the DOM tree. Therefore only element and attribute nodes are involved in making the signature. Let the hash value of the name of a node i be H_i, and the signature be S_i. The S_i is the ORing of all the hash values of its child nodes. That is, the hash value is propagated to its parent node.

Then we can estimate the existence of a certain name l in the sub-tree of the node i by comparison of $H_l \wedge S_i$. If $H_l \wedge S_i \equiv H_l$ then there may be the name l in the sub-tree. Otherwise, if $H_l \wedge S_i \neq S_i$, then we can assure of no existence of the name l in the sub-tree. Table 1 (a) shows hash values of the element and attribute names in Figure 2. Algorithm 1 explains how to calculate the signature of a node, and the results are shown in Table 1 (b).

Algorithm 1 MakeSignature(node)

```
1: s ← 0
2: if node is an Element or Attribute node then
3:     for each ChildNode of node do
4:         s ← s ∨ MakeSignature(ChildNode)
5:         s ← s ∨ Hash(ChildNode.Name)
6:     end for
7: end if
8: node.signature ← s
```

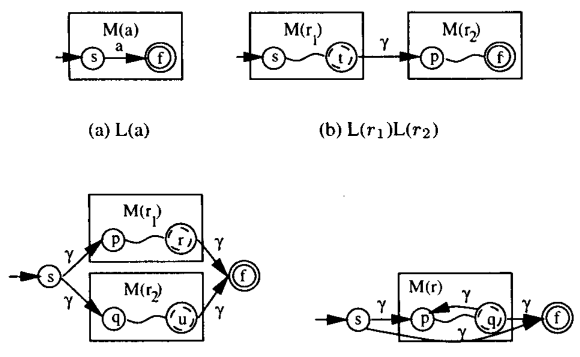
Example 4.1 (Node Traversing) *When we wish to know whether there is a node whose name is* `father` *in the sub-tree of &2 in Figure 2, we perform a bit-wise AND operation between the hash value of* `father`, *$H_{\text{"father"}}$ and the signature of &2, S_2. Since $H_{\text{"father"}} \wedge S_2 \equiv H_{\text{"father"}}$, it is possible that a node whose name is* `father` *exists in the sub-tree of &2. On the contrary, since $H_{\text{"father"}} \wedge S_3 \neq H_{\text{"father"}}$, we can make sure there does not exist such a node in the sub-tree of &3. Therefore, we prune the sub-tree of &3 when finding a node named* `father`.

5. s-NFA(Signature-based NFA)

We propose a scan operator called s-NFA which attaches the signature information to NFA and is used to prune s-DOM as early as possible while traversing s-DOM to evaluate a regular path expression. We will explain how a regular path expression can be transformed to a NFA in Section 5.1. In Section 5.2 we explain how to make s-NFA, and we describe the pruning mechanism in Section 5.3. To avoid confusion of the node in DOM and NFA, we call the node of DOM as an object, and the node of NFA as a state node.

5.1. Query Evaluation using NFA

A regular expression can be represented by an automata. Automata can be deterministic or non-deterministic[18]. A regular path expression is a regular expression as well. In this paper we translate a regular path expression to an NFA.



(a) L(a) (b) L(r_1)L(r_2)

(c) L(r_1 + r_2) (d) L(r*)

Figure 3. NFA

Any complex NFA can be constructed by composition of L(r_1)L(r_2), L($r_1 + r_2$) and L(r*) depicted as Figure 3[18]. L(r?) and L(r+) can be derived by removing certain edge in L(r*).

Definition 5.1 (state set) *The state set is a set of state nodes of NFA, elements are the result of transition in NFA by a certain label path.*

Every regular path expression can be represented as an NFA, and is evaluated by moving the state nodes in NFA while traversing objects in the DOM tree. When we traverse the DOM tree from a given object to its sub-tree, a label path is made. We can create a state set by the label path. If the state set is empty, then query evaluation will be stopped because state transition in NFA cannot have occurred. If a final state node in the NFA is an element of the state set of the object, by which the label path is made, it is accepted as an element of the query result set.

Example 5.1 *The NFA of regular path expression* `AddrList.((person.*)|company).name` *is shown in Figure 4. In this case, any label can be accepted by* `*`, *so* `*` *is the same as* `(any label)*`. *We can obtain a result set* $R = \{$`&4, &18, &7`$\}$ *of this query, which is processed in Figure 2 using Figure 4.*

5.2. s-NFA

State transition in the NFA is determined by the label of the edge. When arriving at the final state by transition, the object in DOM is accepted as an element of the result set. However, we cannot determine which labels appear along the path from the current state node to the final state node. So we have to change state nodes at each step. We have to arrive at the final state node in NFA to accept the objects as a result. Therefore, all labels which come out from the current state node to the final state node must appear when evaluating the queries.

If the labels appearing to the final state node in NFA do not exist in the sub-tree, the objects in the sub-tree cannot be the result of the query, and subsequently, we do not need to traverse that sub-tree. The following definitions are used in making the signature in the NFA.

Definition 5.2 (NFA Path) *The NFA path P_n is a path from a state node n to the final state in an NFA.*

Definition 5.3 (Path Signature) *The path signature PS_n of a state node n in NFA is defined as*

$$PS_n = \{x \mid x \text{ is a value which is ORing hash values of all the labels along a NFA path of state node } n \text{ in NFA}\}$$

The path signature is a bit value which is merged by all hash values of the labels of an NFA path. There are several NFA paths in a state node n because there are several paths from n to the final state node. Therefore, the path signature PS_n of a node n is a set.

The s-NFA proposed in this paper is an NFA of which state nodes have signatures to speed the evaluation of queries. The signatures of the s-NFA are generated by ORing the hash values of all labels that have to be met when moving from the current state to the final state in the NFA. We can examine the existence of the labels that appear from a certain state node n to the final state in the sub-tree of object i in s-DOM. Let the path signature of the state node n be PS_n and the signature of the object i be S_i. Let one element of PS_n be S_n. If $S_n \wedge S_i \equiv S_n$, then we may guess that we can arrive the final state node when traversing the sub-tree of object i. If not, we cannot arrive at the final state node when traversing all objects in the sub-tree of object i. Therefore, we can prune the s-DOM graph by checking the signature.

Figure 3 describes how to build various types of NFA. Therefore, if we can make path signatures of that NFA in Figure 3 then path signatures of any complicated NFA can be built. The rules for making path signatures are described below.

Rule 5.1 (L(a)) *An NFA which has an atomic value as in Figure 3 (a) has a start state s and a final state f. If the hash value of label a is H_a, the path signature of PS_s, PS_f of s and f state nodes, respectively, are*

$$\begin{aligned} PS_s &= \{H_a\} \\ PS_f &= \{0\} \end{aligned}$$

Rule 5.2 (L(r_1 + r_2)) *The path signatures PS_s and PS_f are shown in Figure 3 (c), in which two NFAs are concatenated by* $\vee$.

$$PS_s = PS_p \cup PS_q$$

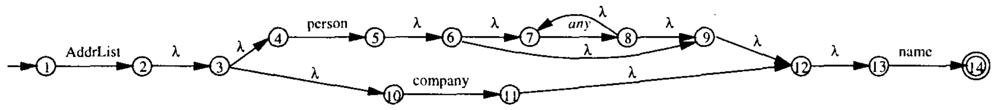


Figure 4. The NFA of `AddrList.((person.*)|company).name`

1	{11101010, 11001001}	8	{10001000}
2	{10101010, 10001001}	9	{10001000}
3	{10101010, 10001001}	10	{10001001}
4	{10101010}	11	{10001000}
5	{10001000}	12	{10001000}
6	{10001000}	13	{10001000}
7	{10001000}	14	{00000000}

Table 2. Path Signatures

$$PS_f = \{0\}$$

Rule 5.3 (L(r*)) *The value of the path signatures PS_s and PS_f of Figure 3 (d), of which operator is *, are 0.*

$$PS_s = \{0\}$$
$$PS_f = \{0\}$$

Rule 5.4 (L(r+)) *L(r+) can be made by removing an edge λ from s to f in Figure 3 (d). Hence, the rules for making path signatures are the same except for PS_s.*

$$PS_s = PS_p$$
$$PS_f = \{0\}$$

The path signature of L(r?) is same as the Rule 5.3.

Rule 5.5 (L(r_1)L(r_2)) *L(r_1)L(r_2) is the concatenation of two NFAs. While we traverse from the start state node to the final state node, the state node p in M(r_2) should be visited. So a path signature PS_i of a state node i in $M(r_1)$ has to be changed by ORing PS_p; that is, $PS_i = PS_i \times_\vee PS_p$. It is the Cartesian product with the path signature of each state node in $M(r_1)$ and PS_p. We call it signature propagation. The path signatures of $M(r_2)$ are not changed. Therefore, the path signature PS_i of each node i in M(r_1) is*

PS_i = { *($x \vee y$) | PS_i is the path signature of a state node in M(r_1), x is an element of PS_i, y is an element of PS_p* }

Example 5.2 (Path Signatures in NFA) *After applying the rules, we can obtain the path signature of each node in s-NFA, and the results are shown in Table 2. For example, PS_{10} is { 10001001 } which is the ORing value between hash values of* `company` *and* `name` *because the edge* `company` *and* `name` *has to be visited in order to arrive at the final state from state node 10.*

Algorithm 2 next()

```
1: /* $S$ is the state set of s-NFA */
2: $node \leftarrow$ get next node by DFS from s-DOM
3: while $node$ is not NULL do
4:    ForwardLabel($S, node$)
5:    ForwardLambda($S, node$) /* using Signature */
6:    if there is a final state in $S$ then
7:       return $node$
8:    end if
9:    if $S$ is empty then
10:      $node \leftarrow$ get next node by DFS from s-DOM
11:   end if
12: end while
```

Algorithm 3 ForwardLambda($S, node$)

```
1: for each state node $n$ which can go forward by $\lambda$ in $S$ do
2:    for each signature $s$ of the path signature of $n$ do
3:       if $s \wedge node$.signature $\equiv s$ then
4:          $m \leftarrow$ the state node moved from $n$ by $\lambda$
5:          add $m$ to $S$
6:          break
7:       end if
8:    end for
9: end for
```

5.3. Query Evaluation using s-NFA

This section describes query processing using s-NFA. The path signature of s-NFA describes what labels have to be visited in order to arrive at the final state from a specific state node in s-NFA. Conversely, the signature of s-DOM shows which labels exist in the sub-tree of a specific object in s-DOM. Before traversing the sub-tree of object o_i in s-DOM we change the state set SS of s-NFA by label l

page size	4K bytes
number of buffer	20
object cache size	500

Table 3. Parameters used in Simulation

of object o_i. When we traverse the s-NFA from one of the state nodes n in SS we compare the signature S_i of object o_i and one of the path signatures PS_n of the state set SS. If $S_i \wedge PS_n \equiv PS_n$ then we can go forward from state node n.

Algorithm 2 is a scan operator that returns a node which is accepted by the regular path expression. The function `next` calls Algorithm 3. In this function, the signatures of s-DOM and path signature of s-NFA are compared to determine whether or not the state of s-NFA can go forward. The meaning of `if` in Algorithm 3 is whether the labels which exist along the current state node to final state in s-NFA exist in the sub-tree of a node in s-DOM. If not, the sub-tree does not need to be visited the remaining sub-tree.

Example 5.3 (Query Evaluation) *When we translate the query of Example 5.1 to s-NFA, the s-NFA can be depicted as similar to the Figure 4, of which each node has a path signature as described in Example 5.2. When object* &1 *is read, state set* $S = \{2\}$. *If we apply Algorithm 3 to progress to states, the labels of which are* λ, *then* $S = \{3\}$ *because the bit operation AND between* S_1 *and 10001001 which is one signature of the path signature* PS_2 *is 10001001. If we apply this operation to object* &2 *then* S *will be* $\{7, 13\}$. *In this situation, AND operation between one signature p of* PS_7 *and* S_5 *can not be p itself. In spite of the query* `person.*`, *the sub tree of* &5 *does not need to be visited. We can obtain results by iterating this operation.*

6. Experimental Results

The simulation program in this paper is coded in Java and evaluates queries in main memory. We store each node of s-DOM as an object and fetch by scan operator, of which the parameter is a regular path expression. The scan operator requests an object from the object cache, which is built on a buffer manager. The object cache requests a page from the buffer manager. The size of each object in the page is not the same for either its length of element name. The object cache and buffer manager use the LRU replacement algorithm. We use two clustering methods, depth-first and breadth-first. The methods are fully clustering algorithms, but real objects may be scattered in the database. We count the number of fetched objects in the object cache and the number of page I/O in the buffer manager. Table 3 shows all parameters used in this paper.

	# of Nodes	File Size
Shakespeare(D_S)	537,621	7.5 Mbytes
Bibliography(D_B)	19,854	247 Kbytes
The Book of Mormon(D_M)	142,751	6.7 Mbytes

Table 4. Characteristics of the XML Files

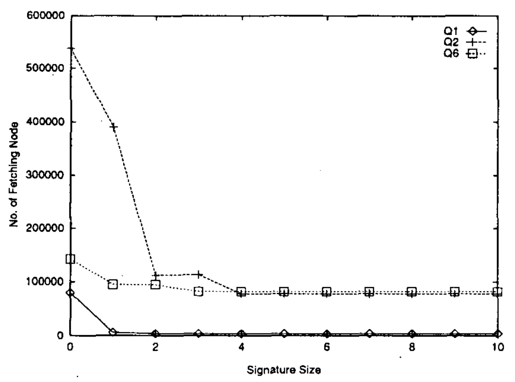
Q1	D_S	PLAY.*[2].PERSONA
Q2	D_S	*.TITLE
Q3	D_B	bibliography.paper.*[1].pages
Q4	D_B	*.author
Q5	D_M	tstmt.*[1].(title|ptitle)
Q6	D_M	*.chapter

Table 5. Queries Used in Simulation

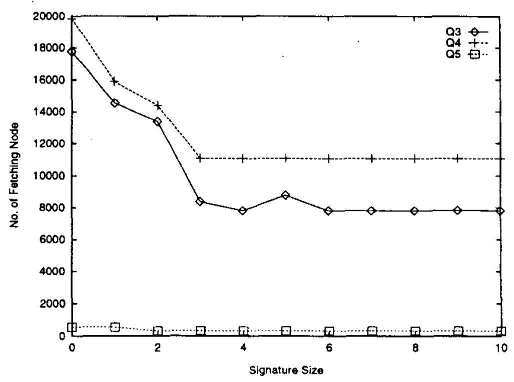
This paper compares clustering mechanisms to determine which is better in traversing the nodes using signatures. Comparing the number of nodes visited and the number of page I/O is the extreme case from the view point of clustering. The number of nodes visited is the performance criterion of a fully unclustered case, while the number of page I/O is that of a fully clustered case. When each node is stored as an object in a database, fetching each object requires a disk operation in the unclustered case. However, when the objects are clustered, fetching each object is not a disk operation. Traversing the tree, several objects near a specific object may be stored on the same page. The clustering methods are BFS and DFS as used in this paper, and the objects are completely clustered. However, after many deletion and insertion operations, objects may be scattered and the clustering status is between clustered and unclustered. In this paper, we show which clustering method is better when signature is used. The data used in this paper are Shakespeare, The Book of Mormon, and the part of Michael Lay's bibliography, which are all translated into XML. The statistics of the data are shown in Table 4.

Six queries were used in the experiment and are described in Table 5. In these queries, *[2] means two paths whose label is an arbitrary string. The first query for each XML data retrieves the data that are located in a specific path. The next query retrieves the data located at any depth of the tree for each data file. Figure 5 shows the results of performance tests. Figures 5 (a) and (b) measure the number of nodes fetched, and (c) and (d) measure the number of page I/O. Queries Q1, Q2 and Q6 fetch many more objects than do queries Q3, Q4 and Q5. Therefore separate graphs are used to distinguish the results. In these figures, zero size of signature means that the signature method is not used.

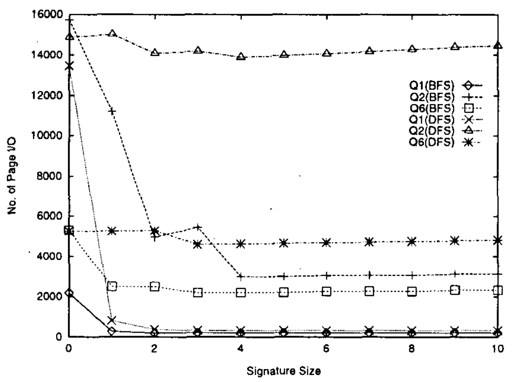
For the number of retrieval of nodes in Figures 5 (a) and (b) the signature-based query evaluation has better perfor-



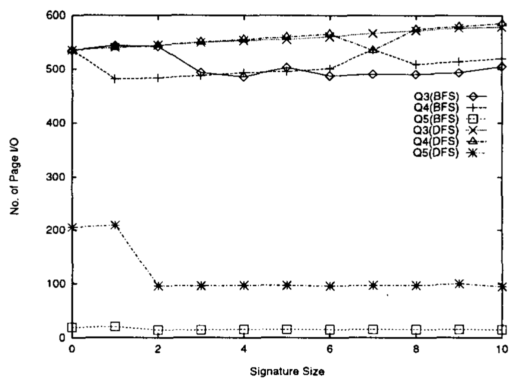
(a) Number of Fetching Nodes (Q1,Q2,Q6)



(b) Number of Fetching Nodes (Q3,Q4,Q5)



(c) Number of Page I/O (Q1,Q2,Q6)



(d) Number of Page I/O (Q3,Q4,Q5)

Figure 5. Performance Evaluation

mance in all cases. This is obtained by decreasing the search space of trees by comparison of signatures between s-DOM and s-NFA. If each node is stored as an object in an object-oriented database, we can decrease the number of objects fetched by the signature method. The larger the signature size, the better the performance. However, when the signature size reaches four bytes, performance improvement ceases. This varies with the number of element names in the XML documents. If the number of element names increases, we have to extend the size of the signature for better performance.

Figures 5 (c) and (d) are the number of disk I/O when XML data is stored as clustering by DFS and BFS. It shows that disk I/O is reduced very significantly in this case. In the general case, we can obtain better performance by BFS. When the query evaluates, the query executor traverses the tree depth-first. However, as s-NFA prunes the sub-tree by the signature method, the possibility is increased of going to a sibling node. In the case of DFS, two sibling nodes may be stored in different pages. Therefore pruning may cause a page fault and a new page is fetched from the database. On the other hand, two sibling nodes may be stored in the same page in BFS. Fetching the sibling node does not cause a page fault in the case of BFS. This is the reason that BFS outperforms DFS. In Figure 5 (d), Q3(DFS) and Q4(DFS) show that the signature method causes more disk I/O. The reason is the overhead of the signature when the objects are stored on disk. The node size of the bibliography is smaller than the other documents when it is stored in the database. In spite of the small size of the signature, there may be a large overhead. However, after many delete and update operations, the nodes cannot be fully clustered and the shape of the graph will be changed as in Figure 5 (b).

7. Conclusion

XML is represented as a tree. When each node is stored as an object in a database, we have to reduce the number of nodes fetched from the database when the queries are evaluated. In this paper we explained the signature method for storing XML documents and evaluating regular path expressions. We can reduce the search space of the graph and disk access by s-DOM and s-NFA. This technique is very useful when an index cannot be used in query processing. The index of semi-structured data is another item of semi-structured data. Therefore, if this technique can be used in a semi-structured index the search space of the index can be reduced.

Clustering is a very important factor for getting better performance. If we cluster the nodes by BFS we can attain better performance than by DFS. That is, clustering between sibling nodes outperforms clustering between parent-child nodes when we use graph traversing based on signature. The reason is that when the graph is pruned in the middle of the graph and a sibling node is traversed, the node may be in the same page when we use BFS.

Acknowledgments

The authors wish to thank Dong-Joo Park for his advice to improve this paper.

References

[1] S. Abiteboul. Querying Semistructured Data. *International Conference on Database Theory*, Jan. 1997.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Library*, 1(1), 4 1997.

[3] S. Abiteboul and V. Vianu. Regular Path Queries with Constraints. *ACM Symposium on Principles of Database Systems*, 1997.

[4] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), 1989.

[5] P. Buneman. Semistructured Data. *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 1997.

[6] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. *SIGMOD*, 1996.

[7] R. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publisher, Inc., 1997.

[8] W. W. Chang and H. J. Schek. A Signature Access Method for the Starburst Database System. *VLDB*, 1989.

[9] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. *SIGMOD*, 1994.

[10] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. *XML-QL: A Query Language for XML*. http://www.w3.org/TR/NOTE-xml-ql, Aug. 1998.

[11] eXcelon. An XML Data Server For Building Enterprise Web Applications. *http://www.odi.com/products/white_papers.html*, 1999.

[12] C. Faloutsos. Signature files: Design and Performance Comparison of Some Signature Extraction Methods. *SIGMOD*, 1985.

[13] M. Fernandez and D. Suciu. Optimizing Regular Path Expression Using Graph Schemas. *ICDE*, 1998.

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

[15] GMD-IPSI. GMD-ISPI XQL Engine. *http://xml.darmstadt.gmd.de/xql*, 2000.

[16] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *VLDB*, 1997.

[17] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. *SIGMOD*, 1992.

[18] P. Linz. *An Introduction to Formal Languages and Automata*. Houghton Mifflin Company, 1990.

[19] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3), 9 1997.

[20] J. McHugh and J. Widom. Query Optimization for XML. *VLDB*, 1999.

[21] A. O. Mendelzon and P. T. Wood. Finding Regular Simple Paths in Graph Databases. *SIAM Journal of Computing*, 24(6), 1995.

[22] T. Milo and D. Suciu. Index Structures for Path Expressions. *ICDT*, 1999.

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

[24] W3C. Document Object Model (DOM). *http://www.w3.org/DOM/*, 2 2000.