ELSEVIER

The Journal of Systems and Software 61 (2002) 91–103

The Journal of Systems and Software

www.elsevier.com/locate/jss

SigDAQ: an enhanced XML query optimization technique ☆

Sangwon Park *, Hyoung-Joo Kim

School of Computer Science and Engineering, Seoul National University, Shillim-dong, Gwanak-gu, Seoul 151-742, South Korea

Received 12 October 2000; received in revised form 5 January 2001; accepted 11 April 2001

Abstract

XML is an emerging standard for data representation and exchange on the Web. XML is represented as a tree and the query as a regular path expression (RPE). The query is evaluated by traversing each node of the tree. Several indexes are proposed for RPEs for fast retrieval. 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 the SigDAQ which is 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. © 2002 Elsevier Science Inc. All rights reserved.

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. To satisfy these requirements, research on applying the previous semistructured data (Abiteboul, 1997; Buneman, 1997) to XML documents has begun. Semistructured data, 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 semistructured data are now broadly applicable to XML (McHugh and Widom, 1999). There are several semistructured or XML database systems, e.g., Lore (McHugh et al., 1997) and eXcelon (eXcelon, 1999).

In the XML data model, an XML document is represented as a graph of which each node is stored as an object in the semistructured database, and queries are evaluated by traversing these nodes. For efficient evaluation of the XML query, decreasing the number of the traversed nodes is important.

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

The above is an example of an XML query, which is similar to Lorel (Abiteboul et al., 1997). This query retrieves person's parents' addresses or telephone numbers. It contains the regular path expressions (RPEs) (Abiteboul et al., 1997; Buneman et al., 1996; Christophides et al., 1994), which are supported by general XML queries such as XML-QL (Deutsch et al., 1998) and XQL. Some syntaxes, such as the star($*$) in XML queries, enlarge the search space when the query is evaluated. 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 (Bertino and Kim, 1989) 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 semistructured data are proposed in Goldman and Widom (1997), McHugh and Widom (1999) and Milo and Suciu (1999) to evaluate the RPEs more rapidly. These indexes have the automata of RPEs and extents to find objects that match given RPEs. The

☆ This work was supported by the Brain Korea 21 Project.
* Corresponding author. Tel.: +82-2-880-1830; fax: +82-2-882-0269.
E-mail addresses: swpark@oopsla.snu.ac.kr (S. Park), hjk@oopsla.snu.ac.kr (H.-J. Kim).

characteristic of these indexes is that they are represented and stored as the same model. For example, the nodes in DataGuide (Goldman and Widom, 1997) are represented as OEM objects; 1-index, 2-index and T-index (Milo and Suciu, 1999), which are semistructured indexes, are represented as semistructured data. Therefore, these indexes are also semistructured data .

The 1-index is for $P.x$, in which P is a regular expression started from the root, whereas 2-index is for $*.x.P.y$. P denotes a regular expression and x, y denote binding variables. This index gives pairs of the start object x and the final object y that match the regular expression P. 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. This index has many nodes, and lots of nodes must be traversed to evaluate the RPE. For this reason the T-index decreases the size of 2-index by reducing the coverage of regular expressions. An example of T-index is $*.person.x.P.y$, and this restricts the start object x, of which parent objects must be matched by $*.person$.

As a result, some data are outside the boundary of these indexes. The path index is for a specific path such as *person.boss.parent*. All possible path indexes cannot be used because of storage requirements. The T-index does not cover all possible RPEs for the same reason. Therefore, we need a new mechanism to evaluate the RPE rapidly when the graph which index cannot cover is traversed. It is also a problem that the index for a semistructured data is another semistructured data. 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.

The data structure of XML documents is represented in DOM (W3C, 2000) as a tree. DOM provides the interfaces for traversing its parent, child and sibling nodes. The tree can be navigated using these interfaces. We propose the SigDAQ (signature-based DOM and query executor) composed of s-DOM and s-NFA which are based on the signature method (Chang and Schek, 1989; Faloutsos, 1985), to reduce the search space when the index is not used for the RPEs. The signature of s-DOM gives a hint as to whether some nodes exist in the subtree of a specific node. The s-NFA is used for evaluating the RPEs using the signature information. This method can be applied to semistructured indexes because they are also represented as a graph. To evaluate the RPEs 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 visits to nodes of the data and index trees.

The size of nodes 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 only a small overhead for computation of the signature.

The method proposed in this paper is used in the optimization module in XWEET (Jeong et al., ngps). XWEET is a system that manages the XML in the Web environment. XML data can be transferred from outside the system, from internal data or produced from other data sources. XWEET is a system that evaluates XML queries from many XML data types available on the Web. XWEET is composed of PDM, wrapper, mediator, XQP (XWEET Query Processor), WPG (Web Page Generator) and HTML/XML Templates.

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

Semistructured data (Buneman, 1997; Abiteboul, 1997) is represented as a graph. The query languages for semistructured data are influenced by those of object-oriented databases such as OQL (Cattell and Barry, 1997) and XSQL (Kifer et al., 1992). 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 semistructured 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, RPEs are used for semistructured queries (Abiteboul et al., 1997; Buneman et al., 1996; Christophides et al., 1994). Indexes of semistructured data (Goldman and Widom, 1997; McHugh and Widom, 1999; Milo and Suciu, 1999) are proposed to execute RPEs 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 RPE.

Theoretical foundations for query processing for semistructured data are studied in Abiteboul and Vianu (1997) and Mendelzon and Wood (1995). Abiteboul and Vianu (1997) uses path constraints for optimization of regular path queries. Fernandez and Suciu (1998) defines a graph schema that has partial information about the graph structure. It reduces the search space by query pruning and query rewriting.

Signature techniques (Chang and Schek, 1989; Yong et al., 1994) are used in database systems. Chang and Schek (1989) used the signature to select matched tuples by select condition. Yong et al. (1994) proposed the technique to reduce page I/O when evaluate the path expression in object-oriented database. However, these methods cannot be used for RPE.

Efficient storage of XML documents to existing database systems has been studied (Deutsch et al., 1999; Florescu and Kossmann, 1999; Shanmugasundaram et al., 1999; Shimura et al., 1999). These methods focus on schema generation to store XML documents in relational databases and on query translation for optimization. There are two methods for storing XML documents in existing database systems. One is by making relational database schemas using the tags of XML documents such as those in Deutsch et al. (1999) and Shanmugasundaram et al., 1999, and each element is stored in the tables. The other is by storing a node of a tree (Florescu and Kossmann, 1999; Shimura et al., 1999) as an object, as XML is represented as a tree.

In the former, XML queries have to be translated to target database languages when we store and extract XML data from the underlying database system, which is represented as a tree and is stored in two-dimensional relational tables. In that case, a wrapper is required to reconstruct the results as XML objects. DTD is important data for making database schemas that are used to store objects. Shanmugasundaram et al. (1999) describes a method for making database schemas to store XML using DTD information. Because many XML documents do not have DTD information, Garofalakis et al. (2000) show a method to extract DTD from data.

The latter method shows that each node of a tree is stored as an object that is used in eXcelon (eXcelon, 1999), PDOM (GMD-IPSI, 2000). The original structure of XML documents cannot be changed by storing each node as an object. Object-oriented databases or Lore (Abiteboul et al., 1997) use this method. We assume that each node in DOM which is the data model of the XML document 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.

3. Data model and query language

The XML data of interest are similar to OEM (Papakonstantinou et al., 1995); that is, a data model of semistructured data. OEM is a self-describing object model and is represented as a graph. The difference between the OEM and XML data model is that the former is represented as a graph, the latter as a tree and ordered list. DOM 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. The sibling nodes are an ordered list.

Fig. 1 is an example of an XML document. This document does not have a DTD that is used to check the validity of an XML document, but DTD is not used in this paper because the databases in this paper store XML documents which have DTD or not. If an XML document does not have a DTD then it is more similar to semistructured data. The DOM structure of this XML document is represented in Fig. 2. The structure is a tree that can be traversed from a certain node to its parent, child or sibling nodes. XML structure can be a graph that has cycles. The method proposed in this paper can be used in cyclic graphs as described in Section 4.3. For simplicity, only trees are used in this paper.

Each node is stored as an object and its OID is represented by '&' as depicted in Fig. 2. For example, the OID of the root node is &1. There are element node, attribute node and text node in DOM, in which each node has a name or a value. The element node and the attribute node have names. The text node is a leaf node and has a value, and the attribute node has a name and a

```
<?xml version="1.0"?>
<!DOCTYPE AddrList>
<AddrList>
  <person>
    <name>
      <first>John</first>
      <last>Smith</last>
    <company>IBM</company>
    </name>
  </person>
  <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>
```

Fig. 1. Example of an XML document.

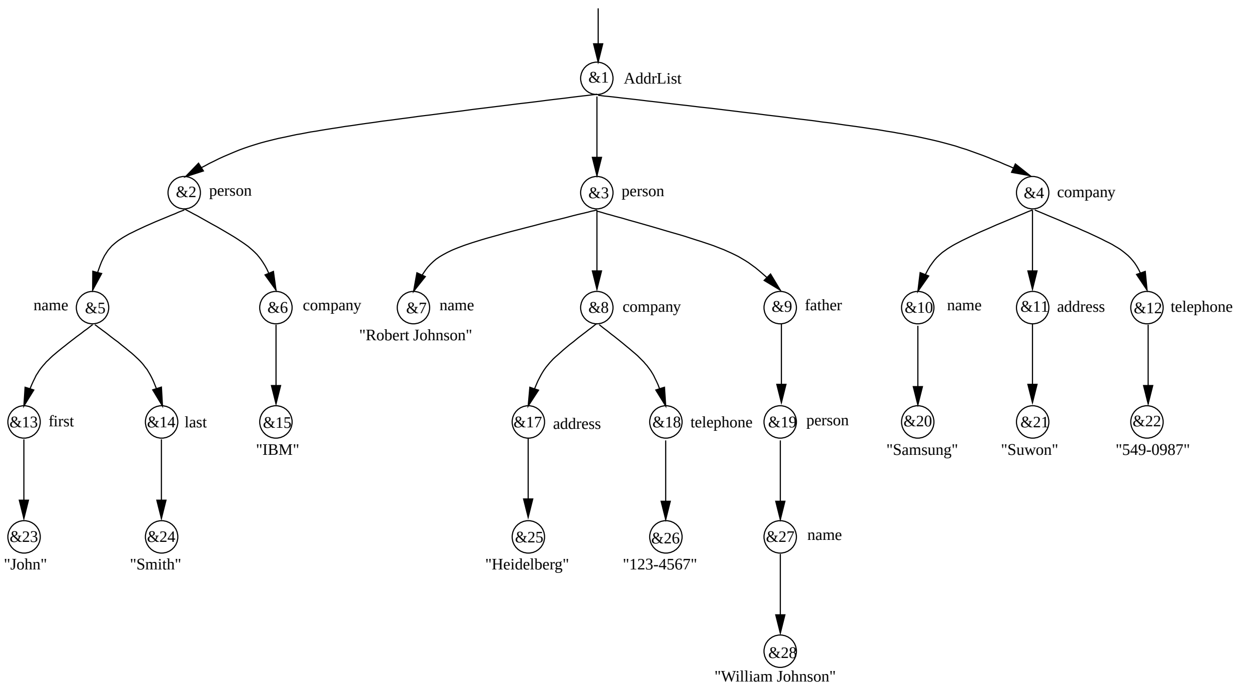


Fig. 2. DOM graph.

value. For example, object &1 and &2 are examples of element nodes whose name is “AttrList”, and object &7 is an attribute node whose name is “name” and value is “Johnson”. The leaf node, such as objects &23 and &24, is a text node. Simple definitions useful for describing the mechanism described in this paper are:

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

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

Queries in this paper are RPEs such as `AddrList.((person.*)|company).name`. They allow wildcard operators such as *, +, ?. The scan operator is provided for searching nodes matched to the given RPE 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 RPEs.

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 section the storage structure, s-DOM, which is based on the signature technique is explained. Each node in s-DOM has not only a node name, but also a signature, that provides a hint for diminishing the search space by pruning worthless graphs as early as possible when an RPE is evaluated. The query processor s-NFA using s-DOM is described in the following section.

4.1. What is signature?

The signature (Chang and Schek, 1989; Faloutsos, 1985; Sacks-Davis et al., 1984) techniques are proposed in the area of full text retrieval. The word signature for a certain string is its hash value generated by a given hashing function like Table 1(a). [1] A given string is tested by the signature whether the string is contained in a certain document block or not. For example, if the three strings, “person”, “company” and “name” in Table 1(a) are stored in the same document block D, the block signature S_D of that document block is obtained by bit-wise ORing all the hash values of the strings in the block, which is $H_{\text{“person”}}$, $H_{\text{“company”}}$ and $H_{\text{“name”}}$:

[1] We make a signature by the SC method of Faloutsos (1985).

Table 1
Hash values of the name of each element and the signatures of each node

(a) *Hash value of string*					
AddrList	01001000	person	00100010		
name	10001000	first	10100000		
last	01000010	company	00001001		
address	01000001	telephone	00101000		
father	00000011				
(b) *Signature of a node in s-DOM*					
&1	11101011	&2	11101011	&3	11101011
&4	11101001	&5	11100010	&6	00000000
&7	00000000	&8	01101001	&9	10101010
&10	00000000	&11	00000000	&12	00000000
&13	00000000	&14	00000000	&17	00000000
&18	00000000	&19	10001000	&27	00000000

$$S_D = 10101011 = (00100010 \vee 00001001 \vee 10001000).$$

Then we can determine whether there is a string "person" in the block D by checking $H_{\text{``person''}}$. If $H_{\text{``person''}} \equiv H_{\text{``person''}} \wedge S_D$, then there is a high possibility that "person" exists in that document block. We call it this *false drop*. However, since for $H_{\text{``address''}}$, the result of $S_D \wedge H_{\text{``address''}}$ is not $H_{\text{``address''}}$. This means that there is no string "address" in that document block. So we can stop searching for a string in that block, thus decreasing the cost of comparison operations.

The property of the signature hashing function used in building s-DOM and s-NFA is that each string value yields m bit positions (not necessarily distinct), in the range 1–F. We can get m and F by the following equations (Faloutsos, 1985). We fix the false drop f as 0.2 and changed the number D which is the number of hash values to make signature. Then we can get the signature size F from 1 to 10 bytes:

$$\log_2 f = -\frac{F}{D \log_2 e},$$

$$m = \text{int}\left[\frac{F}{D \log_2 e}\right].$$

The corresponding bits are set to 1, while all the other bits are set to 0. For example, in Table 1(a), the value "telephone" sets to 1 the bits of positions 3 and 5 ($m = 2$). The signature technique used in this paper is based on superimposed coding (Faloutsos, 1985), that is, ORing of some signatures.

When we search a certain string in the document which is composed of several document blocks, the signature can be used to shrink the search space by comparison between the signature of a document block and the hash value of a given query string. Like the block signature, s-DOM described in the following section, is a method applied signature to DOM in order to shrink the search space when evaluating RPEs.

4.2. s-DOM (signature-based DOM)

In this paper it is assumed that each node of DOM is stored as an object, which is shown in eXcelon (1999), Florescu and Kossmann (1999), McHugh et al. (1997) and Shimura et al. (1999). 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.

As described in the previous section, the existence of a certain name l in the sub-tree of the node i can be estimated by comparison of $H_l \wedge S_i$. If $H_l \wedge S_i \equiv H_l$, then there may be a node whose name is l in the sub-tree. Otherwise, if $H_l \wedge S_i \neq S_i$, then it assures of no existence of the name l in the sub-tree. Table 1(a) shows hash values of the element and attribute names in Fig. 2. Algorithm 1 explains how to calculate the signature of a node, and Example 4.1 shows how to form signatures in s-DOM. The results are shown in Table 1(b), which describes the signature of each node in Fig. 2.

Algorithm 1 (*MakeSignature*(*node*)).

```
1. $s \leftarrow 0$
2. **if** node is an Element or Attribute node **then**
3.    **for** each ChildNode of node **do**
4.       s $\leftarrow$ s $\vee$ MakeSignature(ChildNode) /* bitwise operation */
5.       s $\leftarrow$ s $\vee$ Hash(ChildNode.Name) /* bitwise operation */
6.    **end for**
7. **end if**
8. node.signature $\leftarrow$ s
```

Example 4.1 (*generation of signature*). Let the signature of object &i be S_i and the hash value of the name of object &i be H_i. In this example, it is presented how to generate S_2 of object &2 in Fig. 2. First, S_5 and S_6 should be obtained, since they are the child objects of object &2. The signature S_{13} and S_{14} should be generated for S_5. At this time, S_6, S_{13} and S_{14} are 0 because their child nodes are text nodes. Therefore, S_5 is 11100010 by ORing of signatures $S_{13}(0)$, $S_{14}(0)$ and hash values $H_{13}(10100000)$ and $H_{14}(01000010)$. As a result, S_2 is 11101011 by $S_5(11100010) \vee S_6(0) \vee H_5(10001000) \vee H_6(00001001)$.

In Example 4.1, a parent node has the signature information of its child nodes. This means that the possibility of existence of a node that has a specific name can be determined by comparing the signature between the signature of a node and the hash value of a query string. Example 4.2 shows how the test for the existence of such a node in the sub-tree is performed.

Example 4.2 (*node traversing*). When we wish to know whether there is a node whose name is "address" in the sub-tree of &3 in Fig. 2, we perform a bit-wise AND operation between the hash value of "address", $H_{\text{"address"}}$ and the signature of &3, S_3. If $H_{\text{"address"}} \wedge S_3 \equiv H_{\text{"address"}}$, then it is possible that a node whose name is "address" exists in the sub-tree of &3. On the contrary, for a node whose name is "father", since $H_{\text{"father"}} \wedge S_4 \neq H_{\text{"father"}}$ we can make sure there does not exist such a node in the sub-tree of &4. Therefore, the sub-tree of &4 is pruned when finding a node named "father".

4.3. Signatures of graph

Algorithm 1 needs to be changed when s-DOM is a graph. To make a signature of an object n in s-DOM, we have to obtain the hash values of all object labels of the sub-graph of n. In this case all objects in the graph can be found by graph reachability (Skvarcius and Robinson, 1986). However, the s-NFA explained in the following section can be used unchanged even in cyclic graph.

5. s-NFA (signature-based NFA)

The characteristic of an XML query is that it has RPEs. In this paper the query is evaluated by translating it to non-deterministic finite automata (NFA). The RPE is evaluated by transition of states in NFA. If a certain node in s-DOM arrives to a final state in NFA, the node is pertinent to a query result set. 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 an RPE. It is explained how an RPE can be transformed to an NFA in Section 5.1. In Section 5.2 we explain how to make s-NFA, and 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 which is deterministic or non-deterministic (Linz, 1990). An RPE is a regular expression as well. In this paper an RPE is translated to an NFA. Any complex NFA can be constructed by composition of L(r_1)L(r_2), L($r_1 + r_2$), L(r^*) depicted as in Fig. 3 (Linz, 1990). If the regular expression is `person`, then the NFA is an L(`person`) as in Fig. 3(a). The NFA of `person.name` is a concatenation of L(`person`) and L(`name`); that is,

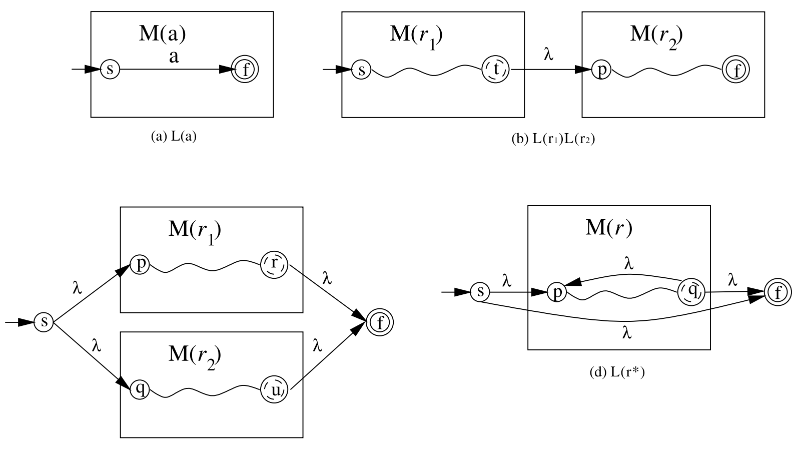


Fig. 3. NFA.

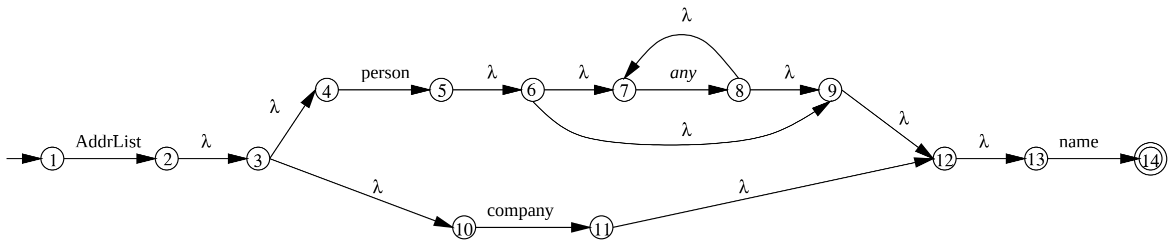


Fig. 4. The NFA of `AddrList.((person.*)|company).name`.

L(`person`)L(`name`) as in Fig. 3(b). An RPE, `person|company`, is an L(`person` + `company`) as in Fig. 3(c), and L(r?), L(r+) is the variation of L(r^*). L(r?) can be derived by removing an edge λ from state q to state p in Fig. 3(d). L(r+) is an L(r^*) in which an edge λ is from state s to state f in Fig. 3(d). The following definition is given to explain the query processing.

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

Every RPE can be represented as an NFA, and is evaluated by moving the state nodes in NFA while traversing objects in the DOM tree. When the DOM tree is traversed from a given object to its sub-tree, a label path is made. If the state set is empty by a given label path, then query evaluation will be stopped because state transition in NFA cannot have occurred again. 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 shows how to make an NFA of an RPE, and Example 5.2 shows the evaluation mechanism using the given NFA.

Example 5.1. The NFA of RPE `AddrList.((person.*)|company).name` is made by Fig. 3, and the result is shown in Fig. 4. In this case, any label can be accepted by `*`, so `*` is the same as `(any label)*`. By merging each NFA, the NFA of the query is constructed with a start state node and a final state node.

Example 5.2. We can obtain a result set $R = \{$`&5, &7, &27, &10`$\}$ of the RPE `AddrList.((person.*)|company).name`, which is processed in Fig. 2. First, the state set SS is {1}, of which an element is the start state node of NFA in Fig. 4. When the root object &1 is fetched from the DOM tree, the label path is `AddrList`, so SS is {4, 10}. Then object &2 is fetched, the label path is `AddrList.person` and SS becomes {7, 13}. After node &5 is fetched, SS becomes {7, 13, 14}. Because the state node 14 is a final state node, object &5 is accepted as the result, ($R = \{\&5\}$). Continuing this operation, the result R of the RPE can be obtained. The tree is traversed by a depth-first search method. [2] In this query, almost every object that is a child object of `person` has to be visited because of `person.*`. This operation will be stopped when SS is empty.

As seen in Example 5.2, if one of the elements of the state set is a final state, then the object is accepted as the result of the query. Otherwise, if the state set is empty, then query processing will be stopped.

5.2. s-NFA

State transition in the NFA in Section 5.1 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 L which come out from the current state node to the final state node must appear when evaluating the queries.

The labels L appearing in NFA should be existed in a sub-tree of DOM. If any label in L does not exist in the sub-tree, the objects in the sub-tree cannot be the result of the query, and subsequently, the sub-tree does not need to be traversed. The following definitions are used in making the signature in the NFA for pruning unnecessary sub-tree.

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

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

The path signature is a bit value which is merged by all hash values of the labels of an NFA path. There are

[2] The tree can be visited by breadth-first search. A queue that stores the nodes is used to visit sub-tree nodes. However, sibling nodes of a node could be much larger, so the queue size cannot be inferred.

several NFA paths in a state node n because there are several paths from the state node n to the final state node. Therefore, the path signature PS_n of a node n is a set. The s-NFA is an NFA of which each state node has a path signature.

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 signature of PS_n be S_j. If $S_j \wedge S_i \equiv S_j$, then we may guess that we can arrive the final state node when traversing the sub-tree of object &i. If not, the final state node cannot be arrived when traversing all objects in the sub-tree of object &i. Therefore, we can prune the s-DOM graph by checking the signature when we are evaluating the queries.

The query is evaluated with the signatures of s-NFA and s-DOM. The signature of s-DOM implies which labels exist in the sub-tree of a specific object and the signature of s-NFA implies which labels have to be met to accept the query. We compare two types of signatures to decide whether to visit a sub-tree or not.

Fig. 3 describes how to build various types of NFA. Therefore, if path signatures of that NFA in Fig. 3 can be made, 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 Fig. 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

$$PS_s = \{H_a\},$$

$$PS_f = \{0\}.$$

Rule 5.2 ($L(r_1 + r_2)$). The path signatures PS_s and PS_f are shown in Fig. 3(c), in which two NFAs are concatenated by union:

$$PS_s = PS_p \cup PS_q,$$

$$PS_f = \{0\}.$$

Rule 5.3 ($L(r^*)$). The values of the path signatures PS_s and PS_f of Fig. 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 Fig. 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 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 traversing 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 ORing with the path signature of each state node in M(r_1) and PS_p. It is called as signature propagation. The path signatures of M(r_2) are not changed. Therefore, the path signature PS_i of each state node i in M(r_1) is

$$PS_i = \{(x \vee y) \mid PS_{i'} \text{ is the path signature of a state node in } \mathrm{M}(r_1),\ x \text{ is an NFA path of } PS_{i'},\ y \text{ is an NFA path of } PS_p\}.$$

Example 5.3 (*path signatures in NFA*). Fig. 5 shows how to apply the above rules to the s-NFA which is made from the query of Example 5.2. The inner box in the

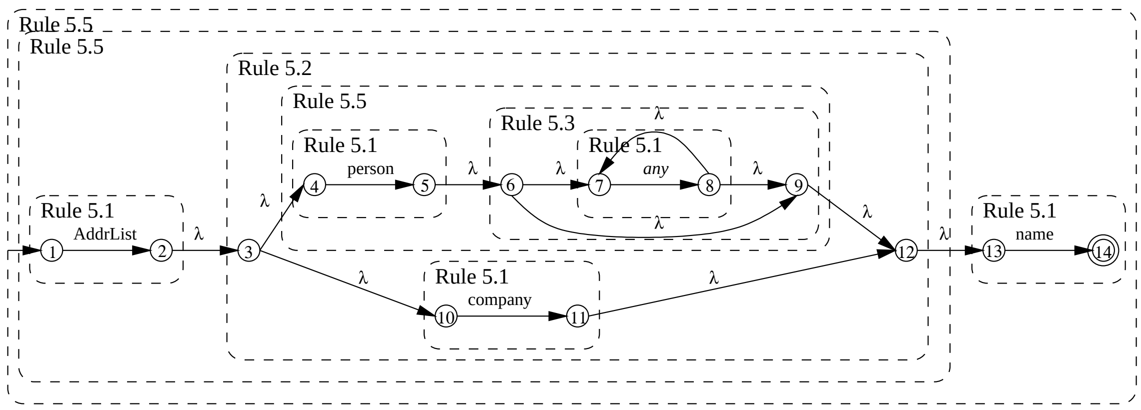


Fig. 5. Applying rules to s-NFA.

Table 2
Path signatures

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

figure is applied ahead of the outer box. After applying the rules, the path signature of each node in s-NFA can be obtained, and the results are shown in Table 2.

For example, if we do not visit edge `name`, we cannot go to the final state from state node 13. As can be seen from Table 2, the path signature PS_{13} of state node 13 in s-NFA is {10001000} which is the hash value of `name`. Because state node 14, the final state, does not have an edge to proceed, then PS_{14} is {00000000}. PS_{11} is the same as PS_{13} because state node 13 should be visited from state node 11 to go to the final state. 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.

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 &*i* in s-DOM, we change the state set *SS* of s-NFA by label *l* of object &*i*. When we traverse the s-NFA from one of the state nodes *n* in *SS*, we compare the signature S_i of object &*i* and one of the signature S_j in PS_n which is path signature of one of the state node in the state set *SS*. If $S_i \wedge S_j \equiv S_j$, then we can go forward from state node *n* and search the sub-tree of &*i*.

Algorithm 2 is a scan operator that returns a node which is accepted by the RPE. 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. The function `ForwardLabel` in Algorithm 2 changes the state set only by the labels of s-NFA without signature information.

Algorithm 2 (*next*( )).

```
1. /* $SS$ 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($SS$, $node$)
5.    ForwardLambda($SS$, $node$) /* using Signature */
6.    if there is a final state in $SS$ then
7.       return $node$
8.    end if
9.    /* sub-tree of $node$ cannot be query result */
10.   if $SS$ is empty then
11.      $node \leftarrow$ get next node by DFS from s-DOM
12.   end if
13. end while
```

The state node *m* in Algorithm 3 is the state node of which a certain signature is S_i. The *node* of s-DOM has the signature S_{node} and $S_i \wedge S_{node} \equiv S_i$. Therefore, it is possible to reach the final state of s-NFA when navigating the sub-tree of *node*.

Example 5.4 (*query evaluation*). When we translate the query of Example 5.2 to s-NFA, the s-NFA can be depicted as in Fig. 4, of which each node has a path signature as described in Example 5.3. When object &1 is read, state set $SS = \{2\}$. If we apply Algorithm 3 to progress to states, the labels of which are λ, then $SS = \{3\}$ because the bit operation AND between the signature of &1 and 10001001 which is one of the 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 of the signature of PS_7 (10001000) and the signature of &5 (11100010) cannot be 10001000. 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.

Algorithm 3 (*ForwardLambda*(*S*, *node*)).

```
1. for each state node $n$ in $SS$, which can go forward by $\lambda$ do
2.   for each signature $S_i$ of $PS_n$ do
3.     if $S_i \wedge S_{node} \equiv S_i$ then
4.       $m \leftarrow$ the state node moved from $n$ by $\lambda$
5.       add $m$ to $SS$
6.       break
7.     end if
8.   end for
9.   remove $n$ from $SS$
10. end for
```

5.3.1. Insert and update

When an object n is inserted or updated in s-DOM, we have to change the signatures of the parent objects of n. The signature of the parent node will be rebuilt by ORing of all labels of child objects of n. The signature is propagated recursively to an upper object until the root is reached. The overheads are greater than for the non-signature method. However, the purpose of this paper is to increase performance by pruning the graph to evaluate the RPE. As the update operation of XML appears infrequently, it is reasonable to ignore the cost of the update operations.

6. Experimental results

The simulation program in this paper is coded in Java and evaluates queries in main memory. Each node of s-DOM is stored as an object and fetched by scan operator, of which the parameter is an RPE. The scan operator requests an object from the object cache, which is built on the 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 its different length of element name. The object cache and buffer manager use the LRU replacement algorithm. Two clustering methods are used in this paper, which are depth-first and breadth-first. The methods are fully clustering algorithms, but real objects may be scattered in the database. Therefore, we shuffle the objects over the database and count the number of page I/O in the buffer manager. Because all operations are executed in main memory, we calculate the disk operation time using the parameters in Ruemmler and Wilkes (1994). The comparison between signatures is bit-wise operation. Therefore, the cost of the CPU time can be ignored, and we just calculate disk operation time. Table 3 shows all parameters used in this paper.

This paper compares clustering mechanisms to determine which is better in traversing the nodes using signatures. Comparing the number of fetched objects and the number of page I/O is the extreme case from the view point of clustering. The number of fetched objects 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 part of Michael Lay's bibliography, which are all translated into XML. The statistics of the data are shown in Table 4.

Six queries are used in the experiment as 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. Fig. 6 shows the number of objects fetched, and Fig. 7 shows the number of page I/O. The time used in disk operations is shown in Fig. 8, which is similar to Fig. 7.

For the number of retrieval of objects in Figs. 6(a) and (b), the signature-based query evaluation has better performance in all cases. 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. The better performance 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

Table 4
Characteristics of the XML files

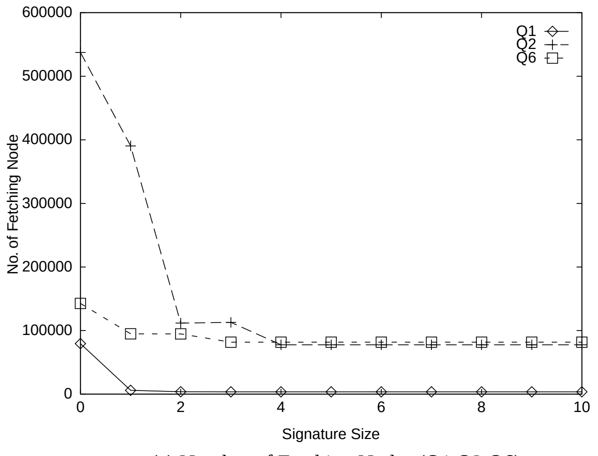
	No. nodes	File size
Shakespeare	537,621	7.5 Mbytes
Bibliography	19,854	247 Kbytes
The Book of Mormon	142,751	6.7 Mbytes

Table 5
Queries used in simulation

Q1	Shakespeare	PLAY.*[2].PERSONA
Q2	Shakespeare	*.TITLE
Q3	Bibliography	bibliography.paper.*[1].pages
Q4	Bibliography	*.author
Q5	The Book of Mormon	tstmt.*[1].(title – ptitle)
Q6	The Book of Mormon	*.chapter

Table 3
Parameters used in simulation

Page size	4K bytes	Number of buffer	20
Object cache size	500	Seek time	$3.45 + 0.597\sqrt{d}$ ms
Sector size	256 bytes	Cylinders	1449
Tracks per cylinder	8	Data sectors per track	113
Revolution speed	4002 rpm	Controller overhead	1.1 ms


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

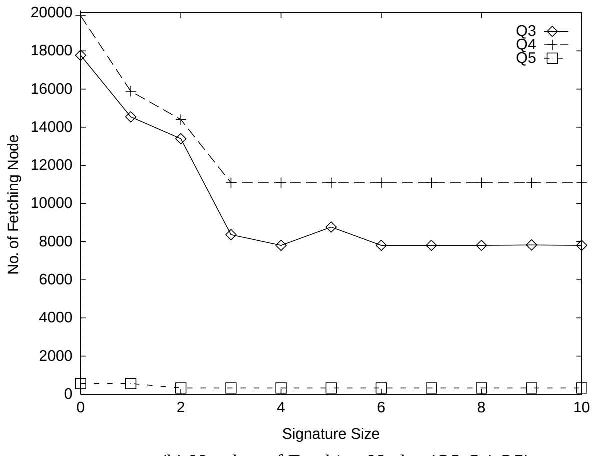
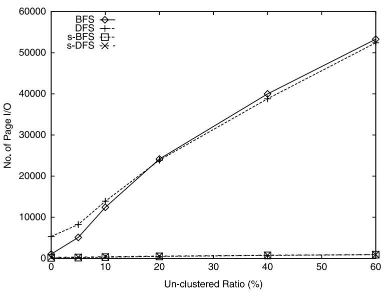
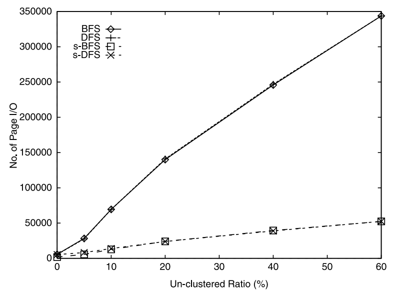
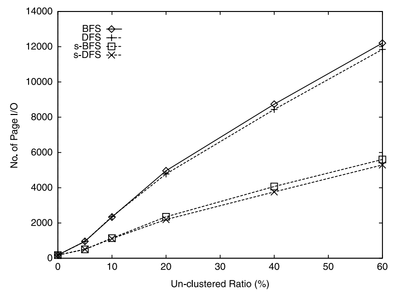
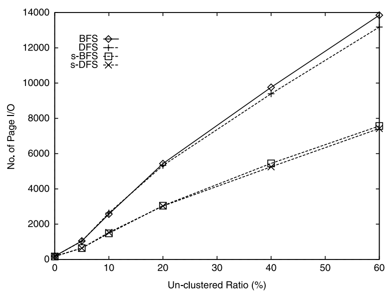

(b) Number of Fetching Nodes (Q3,Q4,Q5)

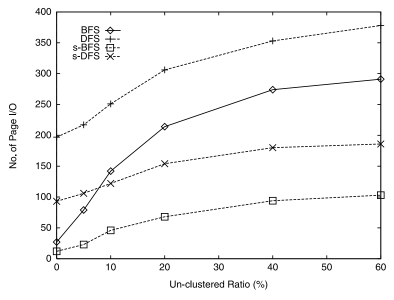
Fig. 6. Performance evaluation.


(a) page I/O(Q1)


(b) page I/O(Q2)


(c) page I/O(Q3)


(d) page I/O(Q4)


(e) page I/O(Q5)

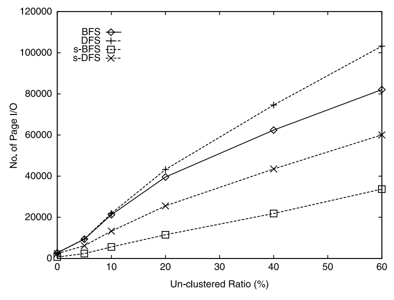
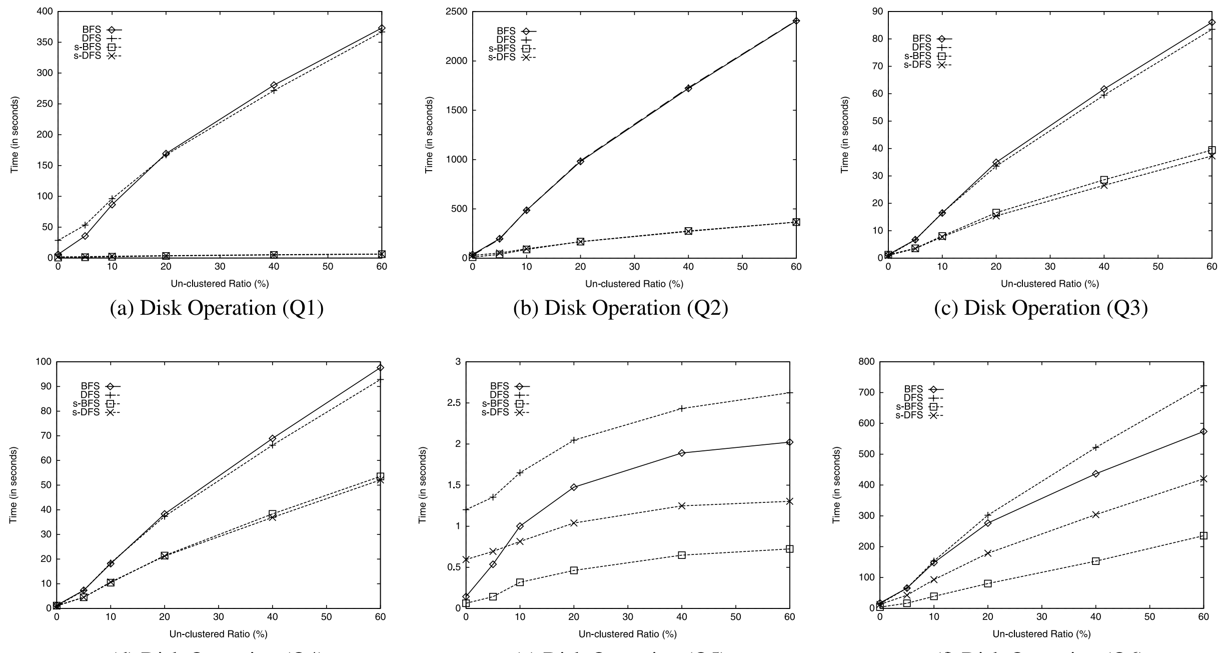

(f) page I/O(Q6)

Fig. 7. Number of page I/O.

repositories, the number of fetched objects can be decreased by the signature method. The larger the signature size, the better the performance. However, when the signature size reaches 4 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 signature for better performance.

Fig. 7 shows the number of disk I/O when XML data are stored as clustering by DFS and BFS and scattering over the pages. It shows that disk I/O is reduced very significantly in each case. In some cases, we can obtain better performance by BFS such as Figs. 7(e) and (f). When the query evaluates, the query executor traverses the tree depth-first. However, as the s-NFA prunes the sub-tree by the signature method, the possibility of going to a sibling node is increased. In the case of DFS, two sibling nodes may be stored in different pages when there are many child nodes. 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. As the objects are shuffled in the object repositories, the



(a) Disk Operation (Q1) (b) Disk Operation (Q2) (c) Disk Operation (Q3)

(d) Disk Operation (Q4) (e) Disk Operation (Q5) (f) Disk Operation (Q6)

Fig. 8. Time of disk operation.

signature method reduces lots of disk I/O. When the objects are scattered over the several pages, fetching an object cause a page fault even it is sibling, parent or child node of the object in the cache. Therefore, the performance gap becomes wider as increasing shuffling ratio.

Fig. 8 shows the calculations of disk operation time. They are very similar to the figures of number of disk I/O. DFS of Q3 and Q4 in this graph are better than BFS, which is different from the others. In spite of decreasing the number of fetched objects, the performance of disk operation is not much improved. The characteristic of the XML document for Q3 and Q4 is that the tree is very flat and not deep. Therefore, when a disk I/O occurs by visiting a child node, the two tracks are more widely separated in BFS than in DFS and thus need more disk operation time.

Clustering is a very important factor for getting better performance when the objects are fully clustered. If the nodes are clustered by BFS we can obtain better performance than by DFS. That is, clustering between sibling nodes outperforms clustering between parent–child nodes when the graph is traversed 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. However, as objects are shuffled, the importance is lessen because page faults are increased.

If the number of nodes are increased or the depth of graph is made greater, then the possibility of saturation is increased. To solve this problem, new technology such as signature chopping is needed. The performance may be improved by using the DTD information of XML.

7. Conclusion and future work

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 SigDAQ which is the signature method for storing XML documents and evaluating RPEs. The search space of the graph and disk access can be reduced by s-DOM and s-NFA. This technique is very useful when an index cannot be used in query processing. The index of semistructured data is another semistructured data. Therefore, if this technique can be used in a semistructured index, the search space of the index can be reduced.

References

Abiteboul, S., 1997. Querying semistructured data. In: International Conference on Database Theory.

Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J., 1997. The Lorel query language for semistructured data. International Journal on Digital Library 1 (1).

Abiteboul, S., Vianu, V., 1997. Regular path queries with constraints. In: ACM Symposium on Principles of Database Systems.

Bertino, E., Kim, W., 1989. Indexing techniques for queries on nested objects. IEEE Transactions on Knowledge and Data Engineering 1 (2).

Buneman, P., 1997. Semistructured data. In: ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.

Buneman, P., Davidson, S., Hillebrand, G., Suciu, D., 1996. A query language and optimization techniques for unstructured data. In: SIGMOD.

Cattell, R., Barry, D.K. (Eds.), 1997. The Object Database Standard: ODMG 2.0. Morgan Kaufmann, Los Altos, CA.

Chang, W.W., Schek, H.J., 1989. A signature access method for the Starburst Database System. In: VLDB.

Christophides, V., Abiteboul, S., Cluet, S., Scholl, M., 1994. From structured documents to novel query facilities. In: SIGMOD.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Suciu, D., 1998. XML-QL: a query language for XML. Available from http://www.w3.org/TR/NOTE-xml-ql.

Deutsch, A., Fernandez, M., Suciu, D., 1999. Storing semistructured data with STORED. In: SIGMOD.

eXcelon 1999. An XML data server for building enterprise Web applications. Available from http://www.odi.com/products/white_papers.html.

Faloutsos, C., 1985. Signature files: design and performance comparison of some signature extraction methods. In: SIGMOD.

Fernandez, M., Suciu, D., 1998. Optimizing regular path expression using graph schemas. In: ICDE.

Florescu, D., Kossmann, D., 1999. Storing and querying XML data using an RDBMS. Data Engineering Bulletin 22 (3).

Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S., Shim, K., 2000. XTRACT: a system for extracting document type descriptors from XML documents. In: SIGMOD.

GMD-IPSI 2000. GMD-ISPI XQL engine. Available from http://xml.darmstadt.gmd.de/xql.

Goldman, R., Widom, J., 1997. DataGuides: enabling query formulation and optimization in semistructured databases. In: VLDB.

Jeong, J.-M., Park, S., Chung, T.-S., Kim, H.-J., 2000. XWEET: XML DBMS for Web environment. In: The First Workshop on Computer Science and Engineering, 2000, Seoul, Korea, pp. 16–17 (2000, http://oopsla.snu.ac.kr/xweet/xweet-eng.ps).

Kifer, M., Kim, W., Sagiv, Y., 1992. Querying object-oriented databases. In: SIGMOD.

Linz, P., 1990. An Introduction to Formal Languages and Automata. Houghton Mifflin, Boston, MA.

McHugh, J., Abiteboul, S., Goldman, R., Quass, D., Widom, J., 1997. Lore: a database management system for semistructured data. SIGMOD Record 26 (3).

McHugh, J., Widom, J., 1999. Query optimization for XML. In: VLDB.

Mendelzon, A.O., Wood, P.T., 1995. Finding regular simple paths in graph databases. SIAM Journal of Computing 24 (6).

Milo, T., Suciu, D., 1999. Index structures for path expressions. In: ICDT.

Papakonstantinou, Y., Garcia-Molina, H., Widom, J., 1995. Object exchange across heterogeneous information sources. In: ICDE.

Ruemmler, C., Wilkes, J., 1994. An introduction to disk drive modeling. IEEE Computer 27 (3).

Sacks-Davis, R., Kent, A., Ramamohanarao, K., 1984. Multikey access methods based on superimposed coding techniques. TODS 12 (4).

Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., Naughton, J., 1999. Relational databases for querying XML documents: limitations and opportunities. In: VLDB.

Shimura, T., Yoshikawa, M., Uemura, S., 1999. Storage and retrieval of XML documents using object-relational databases. In: DEXA.

Skvarcius, R., Robinson, W.B., 1986. Discrete Mathematics with Computer Science Applications. Benjamin/Cummings, Menlo Park, CA.

W3C 2000. Document object model (DOM). Available from http://www.w3.org/DOM/.

Yong, H.-S., Lee, S., Kim, H.-J., 1994. Applying signatures for forward traversal query processing in object-oriented databases. In: ICDE.

Sangwon Park received his BS degree and MS degree in computer engineering from Seoul National University, Seoul, Republic of Korea, in 1994 and 1997, respectively. He is currently enrolled in the Ph.D. program in computer engineering at Seoul National University. His research interests include semistructured data, multidatabase, relational and object-oriented databases.

Hyoung-Joo Kim received his BS degree in computer engineering from Seoul National University, Seoul, Republic of Korea, in 1982 and his MS and Ph.D. in computer engineering from University of Texas at Austin in 1985 and 1988, respectively. He was an assistant professor of Georgia Institute of Technology, and is currently a professor in the Department of Computer Engineering at Seoul National University. His research interests include object-oriented databases, multimedia databases, HCI, and computer-aided software engineering.