IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Hershey PA 17033-1117, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4220

On the Document Storage and Query Evaluation Optimization

Yangjun Chen*
Department of Business Computing, University of Winnipeg, Canada, vchen2@uwinnipeg.ca

Gerald Huck
IPSI Institute, GMD GmbH, Germany, huck@darmstadt.gmd.de

ABSTRACT

Java is a prevailing implementation platform for XML based systems. Several high quality in-memory im-plementations for the standardized XML-DOM API are available. However, persistency support has not been ad-dressed. In this paper, we discuss this problem and intro-duce PDOM (persistent DOM) to accommodate documents as permanent object sets. In addition, we propose a new in-dexing technique: path signatures to speed up the evalua-tion of path-oriented queries against document object sets, which is further enhanced by combining the technique of pat-trees with it to expedite scanning of signatures. Experiments were performed to show that this technique brings really substantial advantages.

INTRODUCTION

Due to the growth of networks, the treatment of electronic information is becoming more and more important. As a subset of SGML, XML is recommended by W3C (World Wide Web Consortium) as a document description metalanguage to exchange and manipulate data and documents on the WWW. The potential of XML is unlimited, and many new applications using XML currently arise, including a Chemical Markup Language for exchanging data about molecules and the Open Financial Exchange for swapping financial data between banks and banks and customers [Bos97]. Also, a growing number of legacy systems are adapted to output data in the form of XML documents.

The Document Object Model (DOM) is a platform- and language-neutral interface for XML. It provides a standard set of objects for representing XML data: a standard model of how these objects can be combined and a standard interface for accessing and manipulating them. There are half a dozen of DOM implementations available for Java from several vendors such as IBM, Sun Microsystems and Oracle. However, all these implementations are designed to work in main memory only. Traditional databases are of limited use for realizing persistency support for the DOM because in many cases XML documents do not adhere to a fixed docu-ment type (even though they are considered to be well formed), from which an efficient database schema can be derived.

In this paper, we introduce a storage method for documents called PDOM (persistent DOM), implemented as a lightweight, transparent persistency memory layer, which does not require the burdensome design of a fixed schema. Great care has been taken to hide implementation details from the application level. As a result, it is possible to plug in a PDOM wherever an in-memory DOM has been used before. A PDOM can replace proprietary database solutions and greatly simplify software design.

In addition, we propose an indexing technique: *path signatures* to speed up the evaluation of queries against documents stored structurally. Using the path signatures, we can avoid traversal along useless paths to expedite path-oriented queries. This technique can be combined with the technique of *pat-trees* [Kn73, Mo68] to make a further improvement of performance. As shown in our experiments (see Section 5), this technique can improve the performance by an order of magnitude or more.

The rest of this paper is organized as follows. In Section 2, we give our system architecture to provide a background for the subsequent discussion. Section 3 is devoted to storage of documents as object sets. In Section 4, we discuss the technique of path signatures and its combination with pat-trees. Section 5 outlines the implementation and reports the experiment results. In Finally, Section 6 is a short conclusion.

SYSTEM ARCHITECTURE

The system architecture can be pictorially depicted as shown in Fig. 1, which consists of three layers: persistent object manager, standard DOM API and specific PDOM API, and application support.

(1) (*persistent object manager*) The PDOM mediates between in-memory DOM object hierarchies and their physical representation in binary random access files. The central component is the persistent object man-ager. It controls the life cycle of objects, serializes multi-threaded method invocations, and synchronizes objects with their file representation. In addition, it contains two sub-components: a cache to improve performance and a commit control to mark recovery points in case of system crashes. These two compo-nents can be controlled by users through tuning parameters.

(2) (*standard DOM API and specific PDOM API*) The standard DOM API methods for object hierarchy ma-nipulation are transparently mapped to physical file operations (read, write, and update). The system aims at hiding the storage layer from an application programmer's view to the greatest possible extent. Thus, for most application, it is sufficient to use only stan-dard DOM methods. The only exception is document creation, which is deliberately left application-specific by the W3C DOM standard. The specific PDOM API allows application to be aware of the PDOM to tune system parameters for the persistent object manager as well as its subsystems: cache and commit control. The specific API is mainly for the fine grained control of the PDOM, not intended for the casual programmers. Rather, it is the place to experi-ment with ideas and proof concepts.

(3) (*application support*) This layer is composed of a set of functions which can be called by an application to read, write, update or retrieve a document. In addition, for a programmer with deep knowledge on PDOM, some functions are available to create a document, to commit an update operation and to com-pact a PDOM file, in which documents are stored as object hierarchies.

In the database (or PDOM pool), the DOM object hierarchies are stored as binary files while the index structures: path signatures are organized as a pat-tree.

STORAGE OF DOCUMENTS AS BINARY FILES

It is organized in node pages, each containing 128 serialized DOM objects. In PDOM, each object (node) corresponds to a document identifier, an element name, an element value, a "Comment" or a "Processing Instruction". The attributes of an element is stored with the corresponding element name together. These object (node) types are equivalent to the node types in XSL [W3C98b] data model. Thus,

* The author is supported by NSERC 239074-01 (242523) (Natural Science and Engineering Council of Canada).

This paper appears in *Issues and Trends of Information Technology Management in Contemporary Organizations,* the proceedings of the Information Resources Management Association International Conference.

Figure 1: Logical architecture of the system

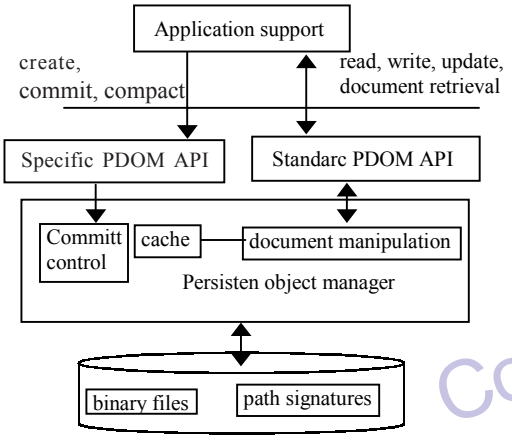
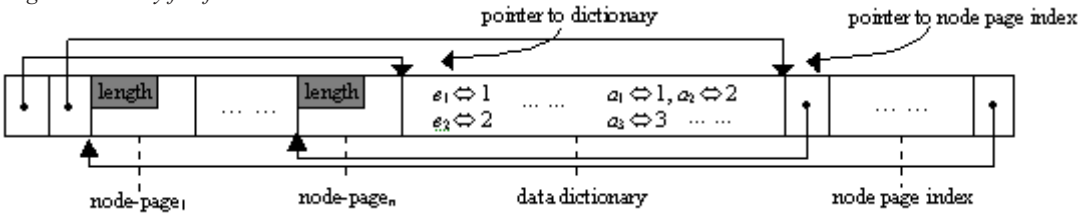


Figure 2: Binary file for documents



a page does not have a fixed length in bytes, but a fixed number of objects it holds. At the beginning of the file, there are two pointers. The first points to a dictionary containing two mappings, by which each element name e_i and attribute a_i are mapped to a different number, respectively; The numerical values are used for compact storage. The second points to the node page index (NPI). The NPI holds an array of point-ers to the start of each node page.

Each object is serialized as follows:

1. A type flag indicating the DOM-type: document identifier, element name, element value, "Comment" or "Processing Instruction".
2. Object content. It may be an integer representing an element name, a PCDATA (more or less comparable to a string), or a string (WTF-8 encoded) representing a "Comment" or a "Processing In-struction".
3. Parent-element identifier (if available).
4. A set of attribute-value pairs. Here, each attribute name is represented by an integer, which can be used to find the corresponding attribute name in the associated data dictionary. The attribute value is a WTF-8 encoded string.
5. Number of sub-elements of an element and its sub-element identifiers.

This serialization approach is self-describing, i.e., depending on the type flag, the serialization structure and the length of the remaining segments can be determined. The mapping between object identifiers in memory (OID) and their physical file location is given by the following equation:

$OID = PI * 128 + i$,

where PI is the index of the containing node page in the NPI and i is the object index within that page. Obviously, this ad-dress does not refer directly to any byte offset in the file or page (which may change over time). Because of this, it can be used as unique, immutable object identifier within a single document. In the case of multiple documents, we associate each OID with a docID, to which it belongs. The following example helps for illustration.

Example 1. In Fig. 3(a), we show a simple XML document. It will be stored in a binary file as shown in Fig. 3(b).

From Fig. 3(b), we can see that the first four bytes are used to store a pointer to the dictionary, in which an element name or an attribute name is mapped to an integer. (For example, the element name "letter" is mapped to "0", "date" is mapped to "1", and so on.) The second four bytes are a pointer to the node page index, which contains only one entry (4 bytes) for this example, pointing to the beginning of the unique node page stored in this file. In this node page, each object (node) begins at a byte which shows the object type. In our implementation, five object types are considered. They are "document", "text" (used for an element value), "element" (for element name), "comments" and 3", "4", respectively. The physical identifier of an object is implicitly implemented as the sequence number of the object appearing within a node page. For example, the physical identifier of the object with the type "document" is "0", the physical identifier of the object for "letter" is "1", and so on. The logic object identifier is calculated using the above sim-ple equation when a node page is loaded into the main memory. At last, we pay attention to the data dictionary structure. In the first line of the data dictionary, the number of the ele-ment names is stored, followed by the sequence of element names. Then, each element name is considered to be mapped implicitly to its sequence number, in which it appears. The same method applies to the mapping for attribute names.

Figure 3: A simple document and its storage

```
<letter filecode="9302">
  <date>January 27, 1993</date>
  <greeting>&salute; Jean Luc,</greeting
  <body>
     <para>How are you doing?</para>
     <para>Isn't it
       <emph>about time</emph>
       you visit?
     </para>
  </body>
  <closing>See you soon,</closing>
  <sig>Genise</sig>
</letter>
```

(a)

byte number		
0:	500	pointer to the data dictionary
4:	565	pointer to the node page index
8:	0	first page number
9:	0	node type "document"
10:	1	number of children
11:	1	integer representing the child's id
12:	2	node type "element name"
13:	0	integer representing "letter"
14:	0	parent ID of this node
15:	1	number of attributes
16:	0	integer representing "filecode"
17:	"9302"	attribute value
22:	5	number of children
23:	2	ID of a child ("date" element)
...	...	...
500:	7	number of element names
501:	letter	an element name "letter"
508:	date	an element name "date"
...	...	...
557	1	number of attribute names
...	...	...
565	8	

(b)

Beside the binary files for storing documents, another main data structure of the PDOM is the file for path signatures used to optimize the query evaluation. To speed up the scanning of the signatures, we organize them into a pat-tree, which reduces the time complexity by an order of magnitude or more. We discuss this technique in the next section in detail.

PATH-ORIENTED LANGUAGE AND PATH SIGNATURES

Now we discuss our indexing technique. To this end, we first outline the path-oriented query language in 4.1, which is nec-essary for the subsequent discussion. Then, the concept of path signatures will be described in 4.2. In 4.3, we will discuss the combination of path signatures and pat-trees, as well as the corresponding algorithm implementation in great detail.

Path-Oriented Language

Several path-oriented language such as XQL [RLS98] and XML-QL [DFF98] have been proposed to manipulate tree-like structures as well as attributes and cross-references of XML documents. XQL is a natural extension to the XSL pattern syntax, providing a concise, understandable notation for pointing to specific elements and for searching nodes with particular characteristics. On the other hand, XML-QL has operations specific to data manipulation such as joins and supports transformations of XML data. XML-QL offers tree-browsing and tree-transformation operators to extract parts of documents to build new documents. XQL separates transformation operation from the query language. To make a transformation, an XQL query is performed first, then the results of the XQL query are fed into XSL [W3C98b] to conduct transformation.

An XQL query is represented by a line command which con-nects element types using path operators (‘/’ or ‘//’). ‘/’ is the child operator which selects from immediate child nodes. ‘//’ is the descendant operator which selects from arbitrary descendant nodes. In addition, symbol ‘@’ precedes attribute names. By using these notations, all paths of tree representation can be expressed by element types, attributes, ‘/’ and ‘@’. Exactly, a simple path can be described by the following Backus-Naur Form:

<simple path> ::= <PathOP> <SimplePathUnit> |
<PathOp><SimplePathUnit>‘@’<AttName>
<PathOp> ::= ‘/’ | ‘//’
<SimplePathUnit>::=<ElementType> |
<ElementType><PathOp><SimplePathUnit>

The following is a simple path-oriented query:

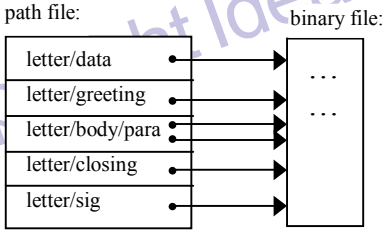
/letter//body [para $contains$‘visit’],

where /letter//body is a path and [para $contains$‘visited’] is a predicate, enquiring whether element “para” contains a word ‘visited’.

Signature and Path Signature

To speed up the evaluation of the path oriented queries, we store all the different paths in a separate file and associate each path with a set of pointers to the positions of the binary file for the documents, where the element value can be reached along the path. See Fig. 4 for illustration.

Figure 4: Illustration for path file



This method can be improved greatly by associating each path with a so-called path signature used to locate a path quickly. In addition, all the path signatures can be organized into a pat-tree, leading to a further improvement of performance.

Signature files are based on the inexact filter. They provide a quick test, which discards many of the nonqualifying values. But the qualifying values definitely pass the test although some values which actually do not satisfy the search requirement may also pass it accidentally. Such values are called “false hits” or “false drops”. The signature of a value is a hash-coded bit string of length k with m bit set to 1, stored in the “signature file” (see [Fa85, Fa92]). The signature of an element containing some values is formed by superimposing the signatures of these values. The following figure depicts the signature generation and and comparison process of an element containing three values, say “SGML”, “database”, and “information”.

When a query arrives, the element signatures (stored in a signature file) are scanned and many nonqualifying elements are discarded. The rest are either checked (so that the “false drops” are discarded) or they are returned to the user as they are. Concretely, a query specifying certain values to be searched for will be transformed into a query signature s_q in the same way as for the elements stored in the database. The query signature is then compared to every element signature in the signature file. Three possible outcomes of the comparison are exemplified in Fig. 3: (1) the element matches the query; that is, for every bit set to 1 in s_q, the cor-responding bit in the element signature s is also set (i.e., $s \wedge s_q = s_q$) and the element contains really the query word; (2) the element doesn’t match the query (i.e., $s \wedge s_q \neq s_q$); and (3) the signature comparison indicates a match but the element in fact doesn’t match the search criteria (false drop). In order to eliminate false drops, the elements must be ex-amined after the element signature signifies a successful match.

Figure 5: Signature generation and comparison

Text: … SGML … databases … information

Representative word signature:		queries:	query Signature	matching results
SGML	010000100110	SGML	010000100110	match with OS
database	100010010100	XML	011000100100	no match with OS
information	010100011000	informatik	110100100000	false drop
object signature (OS)	110110111110			

The purpose of using a signature file is to screen out most of the nonqualifying elements. A signature failing to match the query signature guarantees that the corresponding element can be ignored. Therefore, unnecessary element accesses are prevented. Signature files have a much lower storage overhead and a simple file structure than inverted indexes.

The above filtering idea can be used to support the path-ori-ented queries by establishing path signatures in a similar way. First, we define the concept of tag trees.

Definition 4.1 (*tag trees*) Let d denote a document. A tag tree for d, denoted T_d, is a tree, where there is a node for each tag appearing in d and an edge ($node_a$, $node_b$) if $node_b$ represents a direct sub-element of $node_a$.

Based on the concept of tag trees, we can define path signatures as follows.

Definition 4.2 (*path signature*) Let $root \rightarrow n_1 \rightarrow \ldots \rightarrow n_m$ be a path in a tag tree. Let s_{root}, s_i ($i = 1, \ldots, m$) be the signatures for *root* and n_i ($i = 1, \ldots, m$), respectively.

The path signature of n_m is defined to be $Ps_m = s_{root} \vee s_1 \vee \ldots \vee s_m$.

Example 2 Consider the tree for the document shown in Fig. 3(a). Removing all the leave nodes from it (a leaf always represents

the text of an element) , we will obtain the tag tree for the document shown in Fig. 3(a). If the signatures assigned to 'letter', 'body' and 'pare' are s_{letter} = 011 001 000 101, s_{body} = 001 000 101 110 and s_{para} = 010 001 011 100, respectively, then the path signature for 'para' is $Ps_{para} = s_{letter} \vee s_{body} \vee s_{para}$ = 011001111111.

In the following, we show how to use the path signatures to optimize the query evaluation. As an example, consider the sample path-oriented query given in 4.1 once again.

Example 3 Assume that an additional (hidden) attribute is attached to the relation Element, named *PS* to store path signa-tures for elements. For the path appearing in the query, we first compute its signature:

$s = s_{letter} \vee s_{body} \vee s_{para}$ = 011001111111.

Then, we transform the sample query into the following form:

```
select *
from Element x, Text y
where x.Ename = 'para' and x.PS matches s
        and x.docID = y.docID
        and x.ID = y.parentID and y.value Ê 'visit';
```

where "*matches*" is a function to do the signature checking as described above. From this example, we can see that the path signature can be used to reduce the amount of tuples to be searched in relation Text. It works like a filter to eliminate non-related elements as many as possible. However, due to the "false drops", it is possible that although the path signature of an element matches a query path signature, the corre-sponding path is not the path appearing in the query. Therefore, an extra step is needed to check those paths whose path signature survives the signature checking, which may delay the response time.

Another problem is that some elements may share the same path (e.g., the multiple appearance of "para" element in the tree for the document shown in Fig. 3(a)) and thus the same path signature, which will be redundantly stored as the values of "*PS*" attribute. This problem can be removed by storing all the distinct path signatures in a separate file F_{ps} and establish a hidden attribute (named "pointer") in the relation Element to store the pointers to the positions of the path signatures in F_{ps}. In this setting, the above query can be changed into the following form:

Search the path signature file F_{ps} to find positions whose signature matches *s*;

Let S_{ps} be the set of the resulting positions;

```
select *
from Element x, Text y
where x.Ename = 'para' and x.pointer in Sps
        and x.docID = y.docID
        and x.ID = y.parentID and y.value ⊇ 'visit';
```

To mitigate the first problem mentioned above to some extend, we do not store the path signatures simply in a file, but organize them into a tree structure, the so-called *pat-tree* to find the matching path signatures quickly. We discuss this issue below.

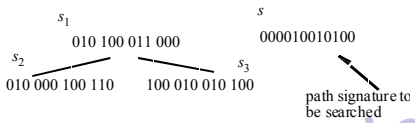
Combining Pat-Trees with Path Signatures

In this subsection, we show how to speed-up the path signature scanning.

As in traditional databases, we want to establish index over path signatures just as a B-tree over a primary key attribute. Unfortunately, due to the fact that signatures work only for an inexact filter, the comparison-and-branching mechanism used in a B-tree can not be utilized to build an index tree structure for signatures. As an counter example, consider the following simple binary tree, which is constructed for an Element relation containing only three tuples.

Assume that *s* = 000010010100 is a signature to be searched. Since $s_1 > s$, the search will go left to s_2. But s_2 does not match *s*. Then, the binary search will return a 'nil' to indicate that *s* can not be found. However, in terms of the definition of the inexact matching, s_3 matches *s*. For this reason, we try another tree structure, the so-called *pat-tree*

Figure 6: A counter example



as the index over path signatures, and change its search strategy in such a way that the behavior of signatures can be modeled. In the following, we discuss how to establish an index for path signatures using pat-trees.

Pat-tree is a digital (binary) tree, by which the key words (or representative words) is represented as a sequence of digits. During a traversal of a pat-tree, the individual bits of the key words are used to decide on the branching.

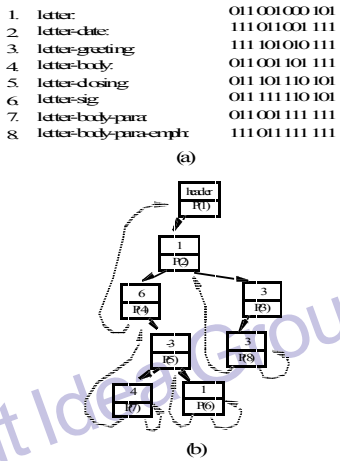
Now we consider each path signature as a digit sequence for a key word and construct a pat-tree for a path signature file as discussed in [Mo68, Kn73].

For example, for the apth signature file shown in Fig. 7(a), we first transform it into a graph shown in Fig. 7(b) (which can then be translated into a tree structure, call a pat-tree; see below)

The graph shown in Fig. 7(a) consists of a header and *n* - 1 = 8 - 1 = 7 nodes. Each node contains six fields:

- Pointer to a signature in the path signature file.

Figure 7: Pat-tree



In Fig. 7(b), P(*x*) (where *x* is an order number of a path signature) shown within each node is a pointer to a path signature, e.g., by P(4), the 4th signature in the path signature file is referred.

- LLINK and RLINK: pointers within the graph. (LLINK is always labeled with 0 and RLINK is always labeled with 1.)
- LTAG and RTAG: one-bit fields which tell whether or not LLINK and RLINK, respectively, are pointers to sons or to ancestors of the node. The dotted arcs in Fig. 7(b) cor-respond to pointers whose TAG bit is 1.
- SKIP: a number which tells how many bits to skip when searching, as explained below. The SKIP fields are shown as numbers within each node of Fig. 7(b).

The graph shown in Fig. 7(b) can be represented as a tree by splitting each node into two ones as shown in Fig. 8. That is, each pointer to a signature is separated from the correspond-ing node. There is an arc from a node *v* to a separated pointer node *u* (corresponding to a pointer to a signature) if there is an ancestor link (dotted arc) from *v* to a node containing *u* in the original graph. In Fig. 8, the node labeled with A will be split into two nodes connected with a dotted line marked with A' and the node labeled with B will be split into two nodes connected with a dotted line marked with B'.

In this way, the graph shown in Fig. 7(b) can be transformed into a tree structure as shown in Fig. 9(a).

Note that in the tree shown in Fig. 9(a), each path from the root to a leaf node corresponds to a prefix of a path signature and the value of that leaf node is its address in the path signature file. Therefore, to check whether a path signature is in a signature file, we simply go along a path and by each node encountered the corresponding bit in that path signature will be checked to decide to go left or right. When a leaf node is reached, the path signature will be checked against the signature pointed by the leaf node.

Figure 8: Node splitting

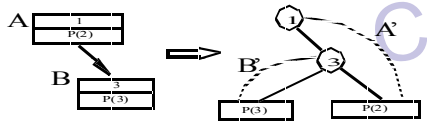
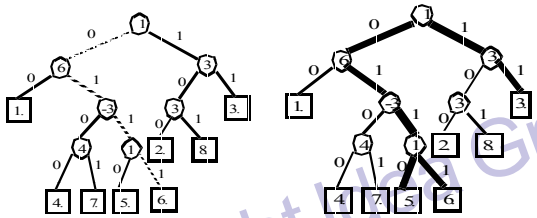


Figure 9: Signatures, path signatures and pat-tree



Note that in the tree shown in Fig. 7(a), each path from the root to a leaf node corresponds to a prefix of a path signature and the value of that leaf node is its address in the path signature file. Therefore, to check whether a path signature is in a signature file, we simply go along a path and by each node encountered the corresponding bit in that path signa-ture will be checked to decide to go left or right. When a leaf node is reached, the path signature will be checked against the signature pointed by the leaf node.

To decide whether some of the signatures in a path signature file "match" a given signature, more time is need. Let s_q be the path signature to be searched. The i-th position of s_q is denoted as $s_q(i)$. During the traversal of a pat-tree, the inexact matching is defined as follows:

(i) Let b be the node (in the pat-tree) encountered and $s_q(i)$ be the position to be checked.
(ii) If $s_q(i) = 1$, we move to the right child of b.
(iii) If $s_q(i) = 0$, both the right and left child of b will be visited.

In fact, this definition just corresponds to the signature matching criterion.

Example 4 Assume s_q = 011 111 110 101. Then, the path marked with dotted edges in Fig. 7(a) will be searched. The bit positions of s_q that will be checked are 1st, 7th (1 + 6 = 7), 4th (1 + 6 - 3 = 4) and 5th (1 + 6 - 3 + 1 = 5), respectively. In the case that s_q = 000 100 100 000, part of the pat-tree (marked with thick edges in Fig. 7(b)) will be searched. On reaching a leaf node, the path signature pointed by the leaf node will be checked against s_q. Obviously, this process is much more efficient than a sequential searching. If the signature file contains N signatures, this method requires only $O(N/2^l)$ comparisons in the worst case, where l represents the number of bits set to 1 in s_q since each bit set to 1 in s_q will prohibit half of a subtree from being visited.

IMPLEMENTATION AND EXPERIMENTS

The path signature concepts have been implemented in a research project aiming at lightweight and efficient XML storage and retrieval in Java. The storage component PDOM[HMF99] stores XML documents in a binary file format which offers random access to individual document nodes. Applications access and manipulate these files via the Java DOM (Document Object Model) API, a standard recommendation of the W3C [W3C98c]. The DOM allows to represent XML documents as a hierarchical structure which consists of different node types representing elements, attributes, or text sections, which offer generic navigation methods and type specific data access and manipulation methods. The PDOM component maps transparently between persistent file data and corresponding DOM objects. It loads and instantiates DOM objects on demand from file, updates the file for modified objects, and caches frequently accessed objects.

The query processor implements XQL as described in [RLS98]. It operates on any Java DOM implementation, but is in addition able to exploit the path signatures provided by our PDOM implementation. Query evaluation is based on a top-down, preorder traversal of the document trees. The top-down approach requires a slightly different path signature encoding. Instead of computing an element's path signature from the parent/ancestor signatures (bottom-up), its path signature is constructed (recursively top-down) from its children/descendants. These signatures allow to prune unnecessary subtree traversals as follows. Before traversing the children of an element, the processor performs a signature test. If the element's path signature matches the query's path signature, the children are visited, otherwise they needn't be considered any further. 17 bits as attribute signature. For simplicity, attribute signatures have not been discussed in the previous sections as they work in the same way as the path signature concept. All the path sig-natures are organised as a pat-tree, stored in a file.

Now we present some benchmark numbers for typical queries against a large XML document. We have compared an OODBMS based XML store (OODB) with XQL query processor, and 'Infonyte', the commercial version of our PDOM and XQL processor, by which the technique described in the previous sections are employed. The tests against Infonyte were performed with enabled cache (IE) and disabled cache (ID). All systems run on a machine with the following configuration:

Dell Poweredge 6300.
4 × Intel Xeon Pentium III (500 MHZ).
1 GB RAM.
4 × 9GB Harddisk.

The OODBMS was configured with 64MB internal in-memory cache, and the Java virtual machine used to run Infonyte is the HotSpot engine that comes with JDK 1.3 from SUN.

The path signatures are stored as longs (64 bits) and thus allow for very efficient signature tests via built-in bit operations on longs. 47 bits of the signature are used as element signature and Jon Bosak. The document uses 7.65 Mbyte file space in textual format. It consists of 180,000 elements and a total number of 327.000 DOM nodes.

The following XQL queries are used for the test:

Q1. /WILLIAM/PLAY/TITLE
Q2. //PLAY/TITLE
Q3. //TITLE
Q4. //LINE
Q5. //PLAY[TITLE="The Tempest"]//
SPEECH[SPEAKER="Lord"]
Q6. //PLAY//INDUCT//SPEECH[//SPEAKER="Lord"]
Q7. //PLAY[//PROLOGUE//SPEAKER="Chorus"]/TITLE
Q8. //PLAY[//INDUCT//SPEECH[//SPEAKER="Lord"]]
/TITLE

The test results are summarized in Table 1.

Queries Q1-Q4 are simple path queries which return <TITLE> elements in <PLAY> elements (Q1, Q2); all <TITLE> elements including <ACT> titles and <SCENE> titles (Q3); and all <Line> elements Q5 returns all <SPEECH> elements where the speaker is "Lord" within the play whose title is "The Tempest", which is in fact an abbreviated form of the following search condition:

//PLAY[TITLE="The Tempest"] and
//PLAY/TITLE//SPEECH[SPEAKER="Lord"]

Table 1: Query execution time (ms.)

Query	OODB	ID	IE
Q1	40	110	<1
Q2	50	280	1
Q3	640	2200	50
Q4	28800	3500	810
Q5	340	310	28
Q6	60	73	3
Q7	1700	230	2
Q8	1800	200	3

Queries Q6-Q8 are more complicated path queries which refer to elements <INDUCT> and <PROLOGUE> that occur infrequently.

Generally, our Infonyte system can compete with the OODB, even with caching disabled. Especially Q4 shows the impact of the cache used by the OODB. The large amount of <LINE> elements (170.000) which are the result of this query do not fit into its cache and lead to an enor-mous performance decrease. With caching enabled, Infonyte outperforms the OODB system at least by a factor of 10.

The execution time for queries Q1-Q4 shows that OODB and Infonyte optimization strategies behave similarly for simple path queries. For the complex path queries Q5-Q8, however, exe-cution time differs significantly. For these queries, our query processor can avoid processing of large, irrelevant subtrees by performing path signature tests, which can seen from Infonyte's uncached evaluation time of queries Q2 (280ms) and Q7 (230ms). Q7 can be thought of as a re-finement of Q2 where the filter '[//PROLOGUE// SPEAKER="Chorus"]' needs to be evaluated for each <PLAY> element's subtree. But before the query processor traverses these trees, it can perform a signature test. This test fails often, as only 5 of 37 <PLAY> elements contain a <PROLOGUE> element. For these remaining 5 elements, the subtree needs to be loaded to evaluate the filter expression. Only 2 <PLAY> elements match this filter expression and are processed further, which finally returns their <TITLE> element.

CONCLUSION

In this paper, a document management system is introduced. First, the system architecture and the document storage strategy have been discussed. Then, a new indexing technique: *path signature* has been proposed to speed up the evaluation of the path-oriented queries. On the one hand, path signatures can be used as a filter to get away non-relevant elements. On the other hand, the technique of pat-trees can be utilised to establish index over them, which make us find relevant signatures quickly. As shown in our experiment, high performance can be achieved using this technique.

REFERENCES

Bos97 Jon Bosak, XML, Java, and the future of the web, March 1997, http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.html.

DFF98 A. Deutsch, M. Fernandez, D. Forescu, A. Levy and D. Suciu, "XML-QL: A Query Language for XML," Aug. 1988, http://www.w3.org/TR/NOTE-xml-ql/.

Fa85 C. Faloutsos, "Access Methods for Text," *ACM Computing Surveys*, 17(1), 1985, pp. 49-74.

Fa92 C. Faloutsos, "Signature Files," in: *Information Retrieval: Data Structures & Algorithms*, edited by W.B. Frakes and R. Baeza-Yates, Prentice Hall, New Jersey, 1992, pp. 44-65.

Kn73 D.E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley Pub. London, 1973.

HMF99 G. Huck, I. Macherius and P. Fankhauser, "PDOM: Lightweight Persistency Support for the Doc-ument Object Model", *OOPSLA'99 Workshop: Java and Databases: Persistence Options,* Nov 1999.

Mo68 Morrison, D.R., PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumer-ic. *Journal of Association for Computing Machinary*, Vol. 15, No. 4, Oct. 1968, pp. 514-534.

RLS98 J. Robie, J. Lapp and D. Schach : "XML Query Language (XQL)," *W3C QL'98 - The Query Lan-guages Workshop*, December 1998.

W3C98a World Wide Web Consortium, Extensible Markup Language (XML) 1.0. http//www.w3.org/TR/ 1998/REC-xml/19980210, Febuary 1998.

W3C98b World Wide Web Consortium, Extensible Style Language (XML) Working Draft, Dec. 1998. http/ /www.w3.org/TR/1998/WD-xsl-19981216.

W3C98c World Wide Web Consortium, "Document Object Model (DOM) Level 1", http://www.w3.org/TR/ REC-DOM-Level-1/, Oct. 1998.