View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Institutional Knowledge at Singapore Management University

Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems   School of Information Systems

4-2003

Efficient Native XML Storage System (ENAXS)

Khin-Myo WIN

Wee-Keong NG

Ee Peng LIM
Singapore Management University, eplim@smu.edu.sg

DOI: https://doi.org/10.1007/3-540-36901-5_6

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Databases and Information Systems Commons, and the Numerical Analysis and Scientific Computing Commons

Citation

WIN, Khin-Myo; NG, Wee-Keong; and LIM, Ee Peng. Efficient Native XML Storage System (ENAXS). (2003). *Fifth Asia Pacific Web Conference*. 2642, 59-70. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/888

This Conference Proceeding Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

ENAXS: Efficient Native XML Storage System*

Khin-Myo Win, Wee-Keong Ng, and Ee-Peng Lim

Centre for Advanced Information Systems, School of Computer Engineering, NTU
Nanyang Avenue, N4-B3C-13, Singapore 639798, SINGAPORE
khinmyo@pmail.ntu.edu.sg, wkn@acm.org, aseplim@ntu.edu.sg

Abstract. XML is a self-describing meta-language and fast emerging as a dominant standard for Web data exchange among various applications. With the tremendous growth of XML documents, an efficient storage system is required to manage them. The conventional databases, which require all data to adhere to an explicitly specified rigid schema, are unable to provide an efficient storage for tree-structured XML documents. A new data model that is specifically designed for XML documents is required. In this paper, we propose a new storage system, named *Efficient Native XML Storage System* (ENAXS), for large and complex XML documents. ENAXS stores all XML documents in its native format to overcome the deficiencies of the conventional databases, achieve optimal storage utilization and support efficient query processing. In addition, we propose a path-based indexing scheme which is embedded in ENAXS for fast data retrieval. We have implemented ENAXS and evaluated its performance with real data sets. Experimental results show the efficiency and scalability of the proposed system in utilizing storage space and executing various types of queries.

1 Introduction

Within a few decades, the Web has been growing incredibly and has become the main information interchange among various organizations. Many applications produce and consume semistructured data which contains irregularities and evolves rapidly making the use of predefined rigid schemas infeasible. XML is emerging as a dominant standard for representing and exchanging semistructured data among applications over the Web. With the tremendous growth of XML data, an efficient storage system is required to manage them.

Several XML storage solutions [1,2,4,6,9,11,12,15] proposed in recent years are based on the conventional databases, such as *relational* (RDBMS) and *object* (ODBMS). The main reason is that these databases are matured enough to handle large volume of data and provide robust data management features. But in practice, they have a lot of limitations to deal with their rigid schema and XML irregular structure. They require an additional transformation to map XML data into their formats and vice-versa. This process is complex and requires more space when the document structure is deep and nested. In addition, they have to

* This work is partially supported by the SingAREN21 research grant M48020004

X. Zhou, Y. Zhang, and M.E. Orlowska (Eds.): APWeb 2003, LNCS 2642, pp. 59–70, 2003.
© Springer-Verlag Berlin Heidelberg 2003

convert XML queries into appropriate patterns understood by underlying query engines. It is time-consuming and performance degradation in data retrieval.

Due to above deficiencies and limitations, a new data model which is specifically designed for XML is emerging recently. This system is able to store XML in its native hierarchical structure and eliminates transformation processes. It provides efficient query processing and document navigation. Currently proposed solutions [8,10,14,17] are designed to provide such advantages but they have not addressed how to efficiently store large collection of documents which contain data with similar hierarchical structure. They disregard the common structural properties of elements, as a result, the storage consumes additional space to maintain redundant structural information and requires more I/O accesses when all elements or contents (values) with a unique path from multiple documents are retrieved.

In this paper, we propose a new schema-conscious efficient native XML storage system, named ENAXS. It organizes a large volume of XML documents with common hierarchical structures and collectively stores elements and contents according to their paths. The new storage aims to eliminate transformation overhead produced in the conventional databases, and overcome inefficient space utilization and query processing of the existing native XML storages. We also embed a new path-based XML indexing scheme into ENAXS to speed up tree traversing and reduce I/O cost for loading and scanning data in query execution.

The rest of the paper is organized as follows. Section 2 reviews related work and Section 3 explores the proposed index structure and storage design for XML data. Section 4 contains the results of our experiments using real data sets and Section 5 concludes the paper.

2 Related Work

Some native XML storage models have been proposed in recent years. *Lore* is an object-based storage that uses *OEM* in which all elements are stored as objects and linked by the use of labels [14]. Lore provides forward traversal but an additional index is needed to traverse backward. *Natix* uses a hybrid approach in which a certain level of data is stored in its structured part and the rest in the flat object part [10]. *Natix* uses an intuitive algorithm to split input document tree into subtrees which are able to be stored in the physical records. This approach provides faster data retrieval when an entire document is needed but inefficient for the query to find all elements with a unique path in multiple documents. *Timber* organizes XML documents as the collection of ordered-labelled trees manipulated by the use of bulk algebra and maps it into *Shore* [8]. *Tamino* is built on the foundation of hierarchical *ADABAS* database [17]. It groups input documents into collections by the definition of an open content model. Each collection contains several document types and each type is assigned a common schema. These schemas are stored in the repository and used in query evaluation.

Several path indexes are also proposed to support faster query processing. *Lore* uses *DataGuides* as a structural summary for document navigation and implements a set of indexes for query execution [7,13]. *ToXin* introduces hash table based indexing scheme that allows efficient traversing along the document [16].

Index Fabric proposes a prefix encoding scheme that encodes all tags along the path and places together with value as a keyword in the Patricia tries [5]. *APEX* uses a structural summary and a hash table to navigate the document and to resolve frequently used path queries [3].

3 ENAXS

We focus to the scenario of storing nodes with a same path expressions together in a group. It may reduce storage space and provides faster data retrieval as most queries expect data with a same structural attribute from multiple documents. ENAXS deals with the approach that all root nodes, internal nodes (elements) and external nodes (contents) from multiple documents are collectively grouped according to their paths, and stored together in the repository. The overall structure of ENAXS is shown in Fig. 1.

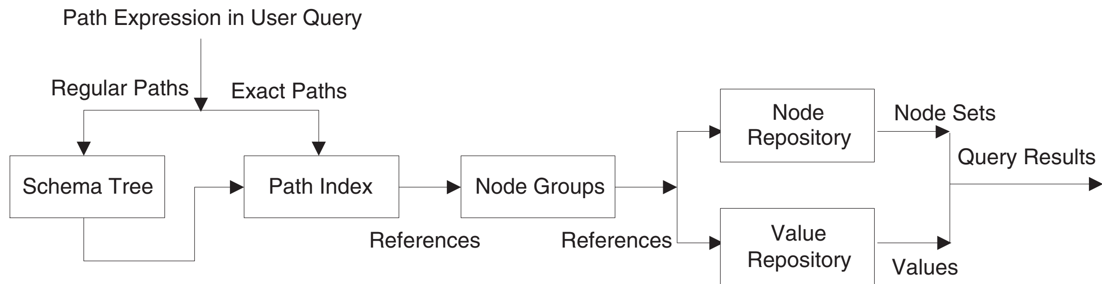

Fig. 1. The Overall Structure of ENAXS

We describe some brief definitions used in ENAXS as follows:

- *Node repository* maintains all root and internal nodes.
- *Value repository* contains binary form of all external nodes.
- *Node group* is a set of node-references grouped together according to their paths regardless of which document they belong.
- *Schema tree* is an abstract representation of the structure of XML documents.
- *Path index* maintains a set of keys which are hash values generated from the path expressions. The leaf nodes of the index tree contain references pointing to the associated node groups.

3.1 Index Structure

A path index is necessary to support queries that requires exhaustive path traversals. With its support, a query processor executes queries without traversing along the given path to fetch the required nodes. The path index in ENAXS comprises of three components: path index, node group and schema tree.

Path Index. Index is uploaded into memory and frequently accessed during query execution, so that, the size, processing efficiency and flexibility of index structure greatly effect on resource utilization and speed of query processing. ENAXS is a path-based storage in which *path index* is embedded as a crucial component by considering above issues. Instead of indexing long strings of path expressions, we generate hash values from it and employ as index keys in order to reduce the overall size of index structure and processing overhead in searching exact-matched string keys. Fig. 2 illustrates the ENAXS index structure.

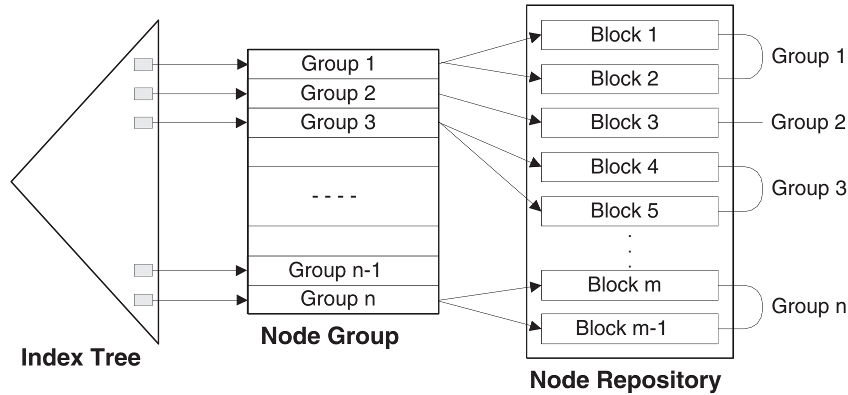

Fig. 2. ENAXS Index Structure

We collectively organize nodes in blocks of physical records and store in the node repository. Each block is addressed by a *block pointer*. We implement the path index using the B^+-tree structure, a balanced tree in which leave nodes are *group pointers* referring to the node groups. Since all index keys in the B^+-tree are numerical hash values, simple arithmetic comparison can be applied in key matching, and the number of B^+-tree levels can be significantly reduced.

Node Group. In order to perform efficient node insertion, update and deletion, we add a layer, named node group, between the index and node repository. Each group in this layer contains references pointing to blocks of nodes in the node repository as depicted in Fig. 2. Since the hash function may generate a same hash key for different paths, ENAXS organizes all path groups and forms a single large one. Fig. 3 shows the general format of a group record.

Each record is composed of *header* and *body*. *Number of groups* in the header indicates the number of node groups in a record, *pointer to group* points to the associated group of node-references and *path expression* consists of a full-path expression that represents a group. As node-references may exceed a block size, *pointer to next block* in the body is used to link an aggregated block and then *group of node-references* is used to point nodes in the node repository. Node insertion, update and deletion in ENAXS become easy because the node group can be updated directly. The following algorithm describes the node insertion procedure.

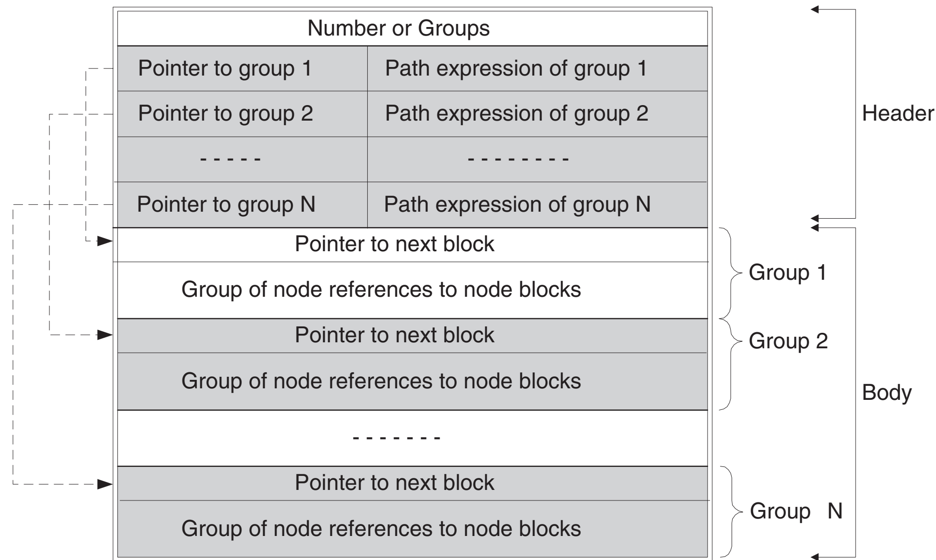

Fig. 3. The Node Group Record Format

Algorithm: Insert(node)

1 **path** ← path expression of node
2 **sig** ← hash(**path**)
3 Search for **sig** in path index
4   If not found, then
5   Store **node** to the node repository in a new block
6   Allocate a new block **new_block** in node group file
7   Create a new node group record, **new_group**, and
            add the reference to **new_block** into **new_group**
8   Insert **sig** and reference to **new_group** into path index
9   else
10   Get the pointer to existing group **old_group**
11   Store **node** into **block** pointed by the last pointer in **old_group**
12   if **block** is full
13   Create **new_block**
14   Store **node** into **new_block**
15   Append **new_block** pointer to **old_group**
16   end if
17   end if
18 end

Schema Tree. In order to deal with regular-path queries, we introduce a structure called *schema tree*, a tree representation of the common structure for a set of XML documents. It is constructed by the use of DTD and provides the parent-child relationship of a pair of nodes. Fig. 4 describes a sample DTD and a XML document and Fig. 5(a) shows its equivalent schema tree.

Each node in the schema tree is assigned a unique identifier so that two nodes from different paths with a same node name can be differentiated. ENAXS

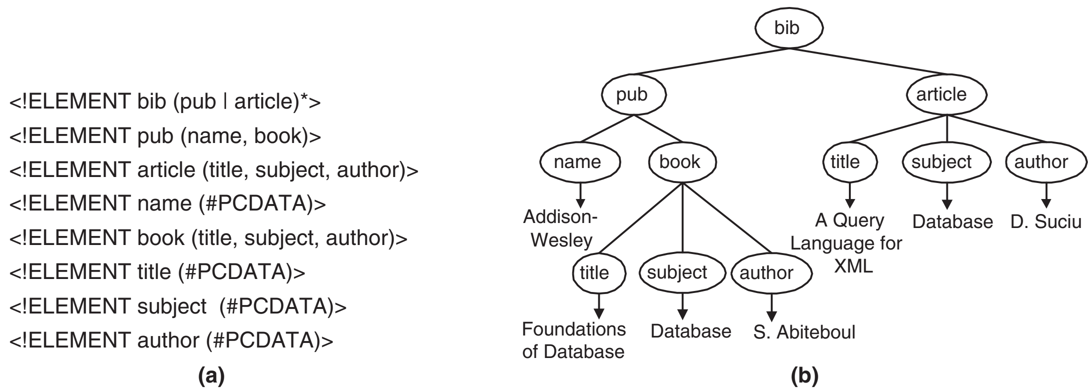

Fig. 4. The Sample DTD and XML Document

traverses along the schema tree to find all possible full-paths that match a given regular path. We employ *bottom-up approach* that enables traversing from the bottom to the root of the schema tree by reducing the possibilities of paths to choose. We implement an *inverted list* illustrated in Fig. 5(b) to support that traversing.

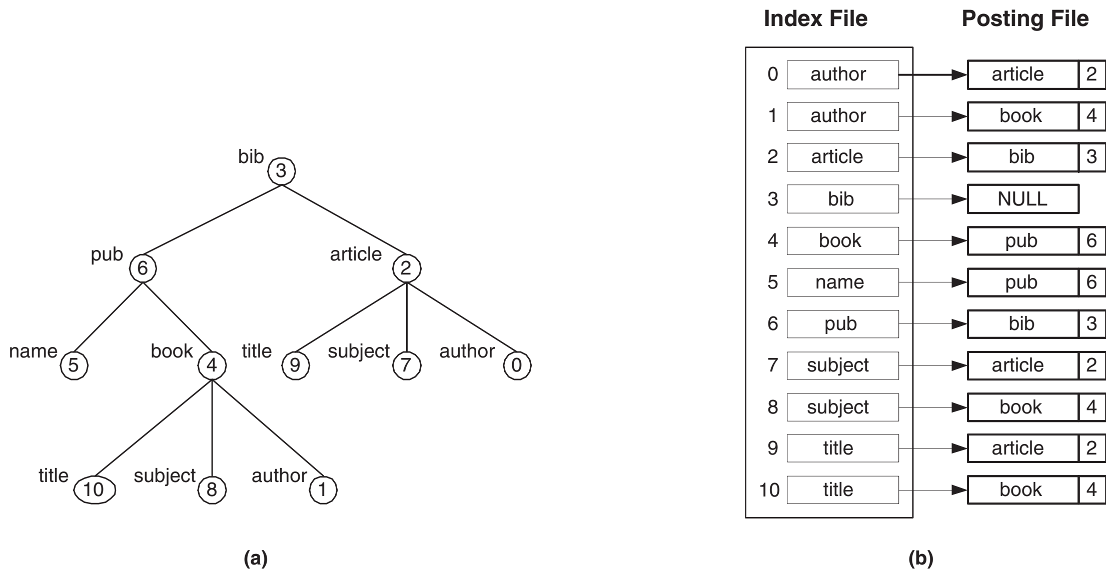

Fig. 5. The Schema Tree and Its Inverted Representation

For a given regular path, we first extract the last tag and then lookup in the index file to locate its parent node in the posting file. We use the resulting node as an index entry to go next level up. This procedure continues till the root node is reached, and all possible full-paths are obtained. Then, the query processor uses the resulting paths to resolve the query. This approach magnifies the capabilities of schema tree not only providing efficient exploring of structure but also

enhancing the query processing because the query can be answered preliminarily by identifying whether a given tag exists in the schema tree, without needing to search in all documents.

3.2 Storage Structure

Typically, internal nodes (root and internal) shape the structure of document and external nodes retain scalar values that are the majority of the document. ENAXS stores structure and contents separately so that the node operations can be performed independently. A unique *document identifier* (DID) is assigned to each document and a *node identifier* (NID) to each node (except external) so that a particular node can be identified by a pair $\langle \texttt{DID}, \texttt{NID} \rangle$ within the storage.

Value Repository. Value repository is a collection of records that are the physical representations of all external nodes. The record format is depicted in Fig. 6(a). An external node record is composed of *DID*, its *Parent Node ID* (PID), *Parent Node Block Pointer* and *Value*. The record size varies with the value length but it is limited by block size. For a record that exceeds a block size, we employ *splitting* to make two parts; the first one is allocated in current block and the rest in a new block, so that documents with long contents such as novels can be stored and the node operations can be performed efficiently.

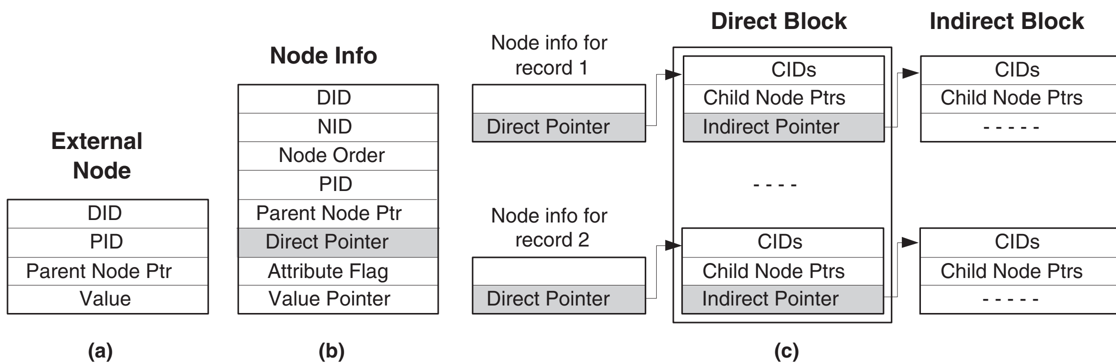

Fig. 6. The External Node and Internal Node Record Format

Insertion of a node is straightforwardly appended to the last block. The *splitting* is applied for a long record to span it over multiple blocks.

Updating of a record invokes the *reallocation* to rearrange all subsequent records by moving up or pushing down within a block. This may acquire a new block if necessary.

Deletion of a node releases space occupied and applies reallocation within the block.

Node Repository. Node repository is a collection of all internal nodes records. (The record format is shown in Fig. 6(b) and (c)). A record is composed of three main parts: node info, direct block and indirect block. A *node info* basically maintains the internal node information that can be used to identify whether it is an element or an attribute and whether it has a value or child nodes. It also maintains a reference that points to its value (value pointer) or a *direct block* (direct pointer) where information of a particular number of child nodes are stored. The additional child nodes, if any, are maintained in an *indirect block*. ENAXS uses a parameter, called the *node set threshold* denoted by α, that predefines the maximum number of child nodes which can be allocated in a direct block. If the number of child nodes is greater than α, indirect pointer is set to reference an indirect block where additional child nodes are stored, otherwise, it is always set to NULL. The value of α is set to 10 by default but it can be fine-tuned by an administrator to achieve the optimal storage utilization. Since node information is fragmented into parts, the node operations can avoid unnecessary loading of vast data.

Insertion of a new node involves creating a new node info, and capturing child nodes information in the direct and indirect blocks or value in the value repository.

Updating of a node simply modifies the node info.

Deletion of a node uses the bottom-up approach that removes value of that node from the value repository or child nodes from direct and indirect blocks. Then it deletes the node info and performs the necessary updating in its parent node.

Our approach improves storage utilization because a direct block can keep several child nodes groups for many internal nodes. It significantly eliminates the storage overhead and reduces expensive I/O accesses in data loading and scanning. Since all node operations modify the structure of documents, it is necessary to update index structure accordingly to reflect the changes.

4 Performance Results

In this section, we present the performance results measured on two aspects: space utilization and I/O cost for storing and querying XML data with ENAXS. We developed the system using Java. IBM XML 4J Parser[1] was used as a parser. The evaluations were performed on a Pentium-III 800MHz machine with 256MB RAM under Windows 2000 using 20GB disk. We use two data sets: Shakespeare's Plays[2] (D1) that contains deep and nested node hierarchy with 327K elements in 37 files (7.6MB in size) and KJV Bible[3] (D2) that is relatively flat with 32K elements in 1 file (4.9MB).

[1] http://www.alphaworks.ibm.com/tech/xml4j
[2] http://www.ibiblio.org/bosak/xml/eg/
[3] http://www.assortedthoughts.com

4.1 Storage Space Utilization

We evaluate the system by altering block sizes (BS) and α to determine how the size of data files (node and value repository, node group and index) reflects the changes. We did not keep track the size of the schema tree as it was small and unaffected by changes in both data sets. Fig. 7(a) shows the growing trend of data files sizes for D1 when BS=4KB and 8KB respectively with α=25, and Fig. 7(b) depicts the situation if α=10.

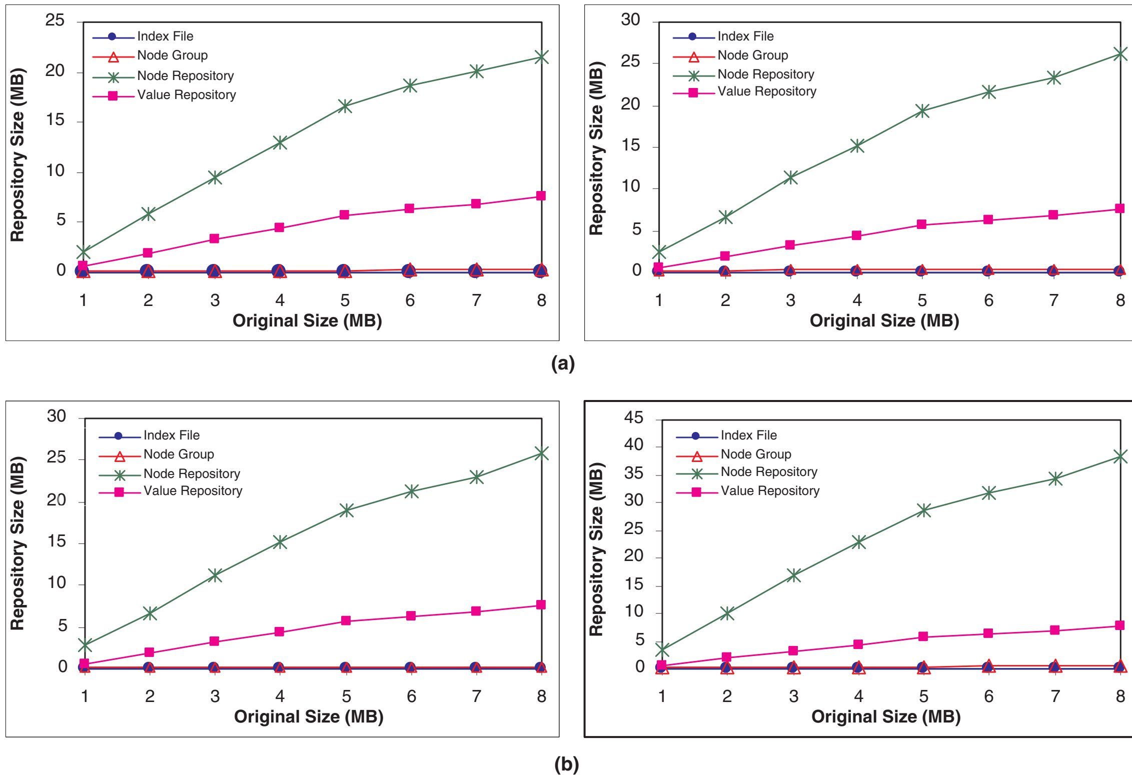

Fig. 7. Size of Data Files

Index File. The path index size is small and the curve is relatively flat no matter how BS changes. It occupies 3 blocks (1 for root and 2 for internal nodes) for D1 as it contains 57 distinct paths and 1 block for D2 as it has only 5.

Node Group File. We found that the size of the node group file increases linearly with respect to data set size. From the experiment with D1, the number of blocks occupied by 57 node groups is 59 when BS is set to 4KB and 57 when BS=8KB. This shows that the number of blocks does not significantly decrease when BS is expanded because each group occupies at least one block. It gives an advantage that there are more spaces available in each block flexible enough

to add more node references when the size of data set increases and reduces I/O accesses for loading fragmented data. We got a similar result for D2.

Node Repository. Node repository is the majority of the storage and it increases linearly with the size of each data set (shown in Fig. 7(a) and (b)). Fig. 8 shows the number of blocks required to store node info, direct blocks and indirect blocks for D1 and D2 respectively. From the several experiments, we observed that ENAXS gives more efficient storage utilization with larger α (25) in both data sets. We also found that larger BS spends less storage space for complex documents that contain large number of nodes, but the smaller BS is better for flat ones with less nodes. For D2 which contains many child nodes for an element, the indirect blocks occupy a large portion of the repository. This indicates that reducing the number of indirect blocks by adjusting α greatly impacts the storage size and support faster execution of node operations.

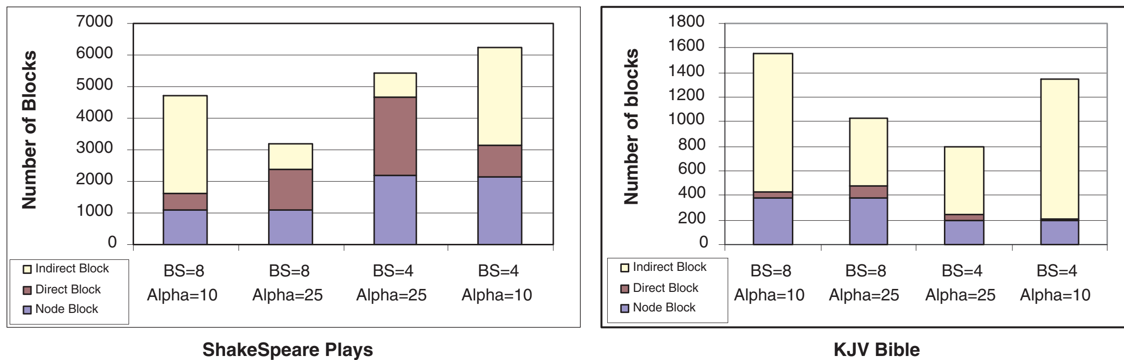

Fig. 8. Block Allocations for Two Data Sets

Value Repository. Value repository is almost the same size as the original data set because values are the majority of both sets.

4.2 Query Performance

We evaluated the performance of ENAXS by executing various queries (described in Fig. 9(a)) using XQuery[4] language on D1 and examined the number of block I/Os accessed during the execution with and without the support of index. We set BS to 8KB and α to 10 and examined the number of blocks read. The results from Fig. 9(b) show that execution of Q1 speeds up approximately 500 times with the support of the index. The execution of Q2 does not cost much than Q1, means that, navigating document with the support of the schema tree spends only a few traversal cost. Q3 and Q4 examine the performance of the value

[4] http://www.w3.org/TR/xquery/

repository. The results show that both queries are substantially faster with the support of the index. Q3 costs more because additional I/Os are needed to access the value repository.

	Query	Description
Q1	PLAY.ACT.SCENE.SPEECH.SPEAKER	Exact-match query to extract all *speaker* in *speech*
Q2	*.SPEAKER	Reqular path query to retrieve all *speaker* in all *play*
Q3	PLAY.ACT.SCENE.TITLE.#	Value query to extract all values of *scene/title*
Q4	PLAY.ACT.SCENE.TITLE	Query to extract all *title*

(a)

	Query	Index	Node Group	Node Repository	Value Repository	Total	No. of Nodes
With Index	Q1	1	1	183	0	185	30972
	Q2	7	6	188	0	201	31067
	Q3	1	1	5	752	759	630
	Q4	1	1	5	0	7	748
Without Index	Q1	0	0	101350	0	101350	30972
	Q2	-	-	-	-	-	-
	Q3	0	0	4658	752	5410	630
	Q4	0	0	4658	0	4658	748

(b)

Fig. 9. Queries and Block I/Os Costs in Query Executions

5 Conclusions

In this paper, we proposed a new schema-conscious native XML storage system called ENAXS to address the shortcoming of maintaining large and complex structured XML documents with similar schema in the existing systems. We take the advantage of the common schema of the XML documents and collectively organize according to their structures so that the system substantially eliminates storage and processing overheads. We also embed a path-based indexing scheme to provide direct access to the nodes. It significantly reduces the number of blocks to be scanned during query processing. With the support of the schema tree, the system can resolve complex regular path queries efficiently.

We conducted various experiments and evaluated the performance of ENAXS in terms of space utilization, I/O assess and query processing costs. The results through experiments have shown that ENAXS supports an efficient and scalable storage for the real XML documents. We will extend ENAXS by adding additional features such as concurrency control, transaction management and multiuser control in our future work.

References

1. D. Alin, F. Mary, and D. Suciu. Storing Semistructured Data with STORED. *SIGMOD Record*, pages 431–442, 1999.
2. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *Proc. ACM SIGMOD Conf.*, Minneapolis, Minnesota, May 1994.
3. C. W. Chung, J. K. Min, and K. Shim. APEX: An Adaptive Path Index for XML Data. *ACM SIGMOD*, 4(6), June 2002.
4. T. S. Chung, S. Park, S. Y. Han, and H. J. Kim. Extracting Object-Oriented Database Schemas from XML DTDs Using Inheritance. In *Proc. 2nd Int. Conf. EC-Web*, Munich, Germany, September 2001.
5. B. F. Cooper, S. Neal, J. F. Michael, R. H. Gisli, and S. Moshe. A Fast Index for Semistructured Data. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pages 341–350, Roma, Italy, 2001.
6. D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDBMS. *In Bullettin of the Technical Committee on Data Engineering*, 22(3):27–34, September 1999.
7. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. 23rd Int. Conf. on Very Large Data Bases*, Athens, Greece, 1997.
8. H. V. Jagadish, Shurug AI-Khalifa, Laks V. S., Andrew Nierman, Stylianos Paparizons, Jignesh Patel, Divesh Srivastava, and Yuqing Wu. TIMBER: A Native XML Database. *VLDB Journal (To appear)*, 2002.
9. S. Jayavel, T. Kristin, H. Gang, Z. Chun, D. David, and N. Jeffrey. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. 25th Int. Conf. on Very Large Data Bases*, Edinburgh, Scotland, 1999.
10. C. C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proc. 16th Int. Conf. on Data Engineering*, San Diego, CA, February 2000.
11. M. Klettke and H. Meyer. XML and Object Relational Database Systems Enchancing Structural Mapping Based on Statistics. In *Int. Workshop on the Web and Database (WebDB)*, Dallas, 2000.
12. K. Loney and G. Koch. *Oracle 8i : The Complete Reference*. McGrawHill, 2000.
13. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3), September 1997.
14. D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo an J. MchHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, and J. Wiener. LORE: A Lightweight Object REpository for Semistructured Data. *ACM SIGMOD*, 25(2):549–549, June 1996.
15. M. Rays. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Bulid Loosely-Coupled Systems. In *Proc. 17th IEEE Int. Conf. on Data Engineering*, Heidelberg, Germany, April 2001.
16. F. Rizzolo and A. Mendelzon. Indexing XML Data with ToXin. In *Proc. 4th Int. Workshop on the Web and Database (in Conjunction with ACM SIGMOD)*, Santa Barbara, CA, May 2001.
17. H. Schoning. Tamino: A DBMS Designed for XML. In *Proc. 17th Int. Conf. on Data Engineering*, pages 149–154, Heidelberg, Germany, April 2001.