Implementation of a canonical native storage for XML

Shihe Ma

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
Ma, Shihe, "Implementation of a canonical native storage for XML" (2004). Retrospective Theses and Dissertations. 20751.
https://lib.dr.iastate.edu/rtd/20751

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Implementation of a canonical native storage for XML

by

Shihe Ma

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Shashi K. Gadia, Major Professor
Daniel Berleant
Xiaoqiu Huang

Iowa State University
Ames, Iowa
2004
Graduate College
Iowa State University

This is to certify that the master’s thesis of

Shihe Ma

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
TABLE OF CONTENTS

ABSTRACT iv

1. INTRODUCTION 1

2. BACKGROUND 3
   2.1 Overview of XML 3
   2.2 Overview of DOM and SAX 3
   2.3 XPath and XQuery 4
   2.4 XML benchmark 5
   2.5 XML databases 6

3. THE CANSTOREX APPROACH 8
   3.1 Architecture 8
   3.2 Disk space manager 9
   3.3 Buffer manager 10
   3.4 ConStoreX DOM API 10
   3.5. XPath query engine 12
   3.6 XML document loading engine 13

4. EXPERIMENTAL RESULTS 18
   4.1 Loading XML documents into CanStoreX 18
   4.2 XPath evaluation 19
   4.3 Comparison of different buffer manager replacement algorithms 20

5. DISCUSSION 22

REFERENCES CITED 24

ACKNOWLEDGEMENTS 27
ABSTRACT

The Extensible Markup Language (XML) is a simple, natural, but powerful language to describe data and metadata and it is being used widely. However, memory limitation becomes the main problem when XML queries are executed against those large XML documents. CanStoreX is a solution for this. This thesis describes the overall design, architecture and implementation of the CanStoreX XML data management system. CanStoreX partitions a large XML document into pages by adding storage-facilitating nodes. As a native XML DBMS, CanStoreX consists of four layers: disk space management, buffer management, CanStoreX DOM API, and XPath query engine. In order to load XML documents into our CanStoreX system, a dynamic bottom-up loading algorithm is proposed. The experimental results show that the CanStoreX system can handle XML data that are up to two orders of magnitude larger than what is currently possible. Some hints are predicted in here to make the CanStoreX storage highly scalable to handle terabyte data.
1. INTRODUCTION

The Extensible Markup Language (XML) is a simple, natural, but powerful language to describe data and metadata [1]. XML is being used widely because it is hierarchical and represents a semi-structured data model. With the growing importance of XML in data exchange, some large repositories of XML data will emerge. In order to retrieve data from XML documents, XQuery has been designed and becoming more popular [2]. Most XQuery processors need to load the whole XML document into the main memory in the DOM (Document Object Model) [3] format, a hierarchical tree structure model, before processing it. However, when a DOM tree is generated, memory requirements will grow nearly linear with the size of the data. In general for a textual document the internal memory requirement is about 7-10 times the size of the UTF8 serialization of this document. This is opposite to the memory requirement in the classical database management system where the memory requirement is only a fraction of the size of the database. Thus, the existing in-memory XQuery implementations are unable to handle large XML documents. Table 1 shows the upper limits of the size of the XML documents that are processed with four popular XQuery programs and two XSLT implementations on an IBM T3 laptop with 256Mb of RAM [4].

Table 1 XML processors maximum document size

<table>
<thead>
<tr>
<th>XQuery Processors</th>
<th>Maximum Document Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quip[5]</td>
<td>7Mb</td>
</tr>
<tr>
<td>Kweelt[6]</td>
<td>17Mb</td>
</tr>
<tr>
<td>IPSI-XQ[7]</td>
<td>27Mb</td>
</tr>
<tr>
<td>Galax[8]</td>
<td>33Mb</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XSLT Processors</th>
<th>Maximum Document Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saxon[9]</td>
<td>50Mb</td>
</tr>
<tr>
<td>Xalan[10]</td>
<td>75Mb</td>
</tr>
</tbody>
</table>
Due to the popularity of XML, large applications are waiting for the XML based usage, but the memory problem is becoming a limitation. The biological data is a typical example of the very large XML documents [11]. Biomolecular Interaction Network Database (BIND) was represented in XML format by Lin [12]. BIND grows fast and its XML representations could reach up to several Giga bytes. Therefore, the efficient approach to store, retrieve, transform and query the BIND XML data is required.

In this paper, we describe CanStoreX, a canonical native XML data management system that breaks an XML document directly into pages. Following that a DOM API is built on the top of CanStoreX and an XPath query engine is implemented using the CanStoreX DOM API.

The page-based organization of the XML document offers the following advantages:

1. CanStoreX mimics the structure of the original XML document. It is a native XML database system.

2. CanStoreX does not require the whole document to be loaded in main memory. Fragments of the document can be processed by reading the pages where these fragments reside. The memory limitation will not be a problem when querying the large XML documents.

3. Benefits similar to the classical relational database management system can be achieved in the CanStoreX system. For example, CanStoreX can read/write a page at a time from/to the disk and process a node at a time by the iterator interface.
2. BACKGROUND

2.1 Overview of XML

XML, a simplified version of Standard Generalized Markup Language (SGML), was announced in 1996 by XML Working Group under the auspices of the W3 consortium. XML is a standard way to delimit text data. The data in XML is also self-describing. XML documents can be either document-centric or data-centric. Document-centric design involves a liberal use of free-form text that is "marked up" with elements. This design reflects the origins of XML from SGML. Document-centric documents are easy to render on some sort of output device. A book and HTML are typical examples for the document-centric XML. On the other hand, data-centric documents are typically easier to process with computer programs because the data is better organized. Of course, unlike the document-centric document, the order of nodes is not important for the data-centric XML document. Most XML documents are typically data-centric.

2.2 Overview of DOM and SAX

There are two most widely used APIs for working with XML documents. One is the Simple API for XML (SAX) [13]. The other one is the Document Object Model (DOM) [3]. The W3C maintains the Document Object Model (DOM) Recommendation. A DOM parser reads the entire document, constructs a tree of objects in memory, and provides a tree-structured view of the XML document. The major component structures of the document are nodes in the objected tree. Navigation of the DOM tree can be done by the DOM interfaces.
The W3C DOM specification only provides the interface definition for the DOM library not the specifics of their implementation. Therefore, the DOM implementation is left open. Several implementations are available. One of the most widely used DOM implementations is Xerces from Apache XML Project that is implemented with Java. It is used in our system.

The following functions of a W3C DOM node are important for navigating a DOM tree: getParentNode(), getChildNodes(), getFirstChild(), getLastChild(), getPreviousSibling, and getNextSibling. Unlike the DOM, SAX is not the product of a standards organization. SAX becomes popular because of its different approach to access XML documents. SAX provides events sequentially instead of a tree view when the document is parsed. For every element, two events can be triggered: the start event and the end event. For the text node (leaf node), only one event is triggered. SAX parser processes one event at a time so that the minimal memory is required. It is ideal for handling very large XML documents. However, the SAX does have some drawbacks. SAX can not randomly access to the document so it is read-only and not updatable. The complicated search can not be implemented using SAX so that most XQuery engines have to be built on top of the DOM implementations.

2.3 XPath and XQuery

XML is used as a storage model for representation and exchange of information among various applications. It is natural to expect that an ideal query language to retrieve desired information from XML documents becomes necessary. XPath and XQuery satisfy these needs.

XML Path Language (XPath) is a set of syntax rules for defining/addressing parts of an XML document [14]. In addition to its use for addressing, XPath is also used to test
whether or not a node matches a pattern. XPath expressions are similar to the file path expressions in a computer file system. XPath is a major element of the XSLT standard.

XQuery is designed by XML Query and XSL Working Groups [2]. It is derived from Quilt, which itself borrowed some features from several other languages including XPath 1.0 [14], XQL [15], XML-QL [16], SQL and OQL [17]. The newer XQuery 1.0 is a super set of XPath Version 2.0. The basic introduction of XQuery could be found in XML Query (XQuery) website [2]. There are several other documents on W3C website that introduces different aspects of XQuery [18]. The XQuery project includes not only the standard for querying XML documents, but also the next-generation standards, such as XML selection (XPath2), XML serialization, full-text search, functional XML data model, and possible functions and XML Data Model, and a standard set of functions and operators for manipulating web data.

2.4 XML benchmark

With the development of XML databases, the benchmark framework has become more and more important and necessary. XMark is one of the widely used benchmarks for XML databases [19]. XMark data generator produces XML documents modeling an auction website, a typical e-commerce application. The generated XML document contains well-formed, valid, and meaningful XML data. The file size of the generated XML document is scalable and could be up to several Giga bytes. For example, if the scaling factor is 1.0, the output file size will be about 100MB. The sample data taken from the relevant portion of an XMark document is shown in Figure 1.
2.5 XML Databases

XML documents are often stored in a file system as text files. Storing large XML documents as text files is impractical. The increasing volume of available XML data is spurred the development of XML Database Management Systems to allow users and applications to query large stores of semi-structured data. Researchers have developed a great

Figure 1 Portion of Xmark databases
number of XML database management systems. They fall into two main categories. The first category of XML databases is to map and store XML documents to an existing DBMS, such as relational/object DBMS, then to retrieve or query the data from the existing database by SQL/OQL, or to support XQuery by executing the SQL/OQL [20, 21, 22, 23, 24, 25, 26, 27, 28]. In the last few years, various strategies for mapping XML data to relational database system have been proposed. These strategies includes: edge table [29], attribute table [29], universal table [29], inline [29], full shredding [30], dynamic interval encoding [31, 32], and XParent [33]. These strategies can be categorized into two groups: edge-oriented and node-oriented. Some strategies may lose the order of the XML document and can not restore the original tree structure. All the above products on top of existing DBMS support one or more XML query languages. Also some XML query languages have been proposed to update the XML database in some products based on RDBMS [34, 35]. The second category of XML databases is the native XML database. Timber is a native XML database product developed by University of Michigan [36]. The Timber system is based upon a bulk algebra for manipulating trees, and natively store XML on top of the Shore, a persistent object system. Natix is another native XML database product implemented by Software AG [37]. Natix splits the XML document into small records using proxy nodes. Records are then stored in disk blocks. XQuery languages are supported by the above two native XML database management systems. In addition, Persistent DOM may be considered as a type of native XML database [38].
3. THE CANSTOREX APPROACH

CanStoreX breaks an XML document directly into pages. Each page itself is organized as a legal XML document and page-based organization of the document directly mimics the structure of the original document. The recursive largeness in an XML element is due to two reasons: the largeness of fanout or largeness of individual children. To paginate the large XML document tree, two types of storage-facilitating nodes are added to an XML document: \textit{f-nodes} and \textit{c-nodes}. The \textit{f-node} is added to group a sequence of one or more children of a parent together. The \textit{c-node} contains a page ID pointing to a child page where a subtree rooted at an \textit{f-node}. After the large XML document tree is partitioned into pages by adding some necessary storage-facilitating nodes, the resulted pages can be stored on the hard disk directly.

3.1 Architecture

The overall architecture of CanStoreX is shown in Figure 2. We built our XML data management system from the scratch using Java which includes four layers: disk space management, buffer management, CanStoreX DOM API, and XPath query engine.
3.2 Disk Space Manager

The disk space manager layer supports the concept of a page as a unit of data, and provides commands to allocate or deallocate a page and read or write a page. The pages are stored as disk blocks so that reading and writing a page can be done in one disk Input/Output (I/O). In our research, the OS file systems are used to manage disk space. A Java random accessed file is created and a number of blocks are allocated and initialized. Then, the whole XML document can be paginated and loaded into these blocks.
3.3 Buffer Manager

The buffer manager layer is responsible for bringing pages from disk to main memory as needed. The collection of main memory frames used by the buffer manager for this purpose is called the buffer pool. The frames in buffer pool have the same size as the pages on the disk. In our system, the page size and frame size are 4KB, respectively, and the buffer pool contains 100 frames. If a requested page is not in the pool and the pool is full, the buffer manager’s replacement policy controls which existing page is replaced. Six replacement policies are being compared in our system. The used replacement algorithms include: first in first out (FIFO), least recently used (LRU), most recently used (MRU), least frequently used (LFU), most frequently used (MFU), and random replacement (Random). The buffer manager keeps track of the counters for the requested pages and the disk block accesses. Usually, a database system is fast if the number of blocks that are read from memory is maximized. The term that is related to that idea is “Hit Ratio”, defined as 100% * (total blocks read from memory) / (total blocks read). This is the same as 100% * (1 - (blocks read from disk)/(total blocks read)).

3.4 ConStoreX DOM API

The data in every page actually is a legal XML document. The small document is represented in memory as a small DOM tree. All small DOM trees put together to form a large virtual DOM tree. Our ConStoreX DOM API is built on the top of the virtual DOM tree. These small DOM trees are hidden from users and ConStoreX DOM API has the same behavior as the classic memory-based W3C DOM API. The following sections describe the main algorithms for the DOM API functions.
getParentNode: If the parent node is in the same page as the given node, and the parent node is not an f-node, the parent node is the desired node. If the parent node is an f-node, follow the pointer provided by the f-node to find the parent node.

defineFirstChild: The first child node is the leftmost child node of the current node. If the first child node is in the same page as the given node, and the first child node is not a c-node, the child node is the desired node. If the child node is a c-node, follow the pointer provided by the c-node to find the first child node.

defineLastChild: The last child node is the rightmost child node of the current node. The algorithm to find the last child node is similar to the one of defineFirstChild.

definePreviousSibling: If the previous sibling node is in the same page as the given node, and the previous sibling node is not a c-node, the sibling node is the desired node. If the sibling node is a c-node, follow the pointer provided by the c-node to find the f-node in the child page, and the last child node of the f-node is the desired node. If no previous sibling node is found in the current page and the parent of the current node is an f-node, follow the pointer provided by the f-node to find the c-node in the parent page, the previous sibling node of the c-node is the desired node.

defineNextSibling: The algorithm to find the next sibling node is similar to the one of definePreviousSibling.

defineAttributes: The attributes associated with the element node are kept in the same page. Therefore, the attributes can be directly returned from the small DOM tree in a single page. For the element containing a very large number of attributes that cannot fit in a single page, we can split the attributes into different pages by some other type of storage-facilitating node.
**getChildNodes:** The in-memory DOM implementation materializes all child nodes at once, which is not a suitable strategy in database management. In order not to materialize the all child nodes, the iterator interface is introduced here which supports pipelining of results naturally. The iterator interface includes the functions: open, getNext, and close. The open function initializes the state of the iterator by putting the first child node in the buffer. The getNext returns the node in the buffer and calls the getNextSibling function to replace the buffer with the next sibling node. When the last child is returned from the buffer, the close function is called to deallocate state information.

### 3.5 XPath Query Engine

An example of an XPath expression is as follows:

```xml
document("xmark.xml")//item[location/text()='United States']/mailbox
```

The above XPath expression returns the mailbox information of all items containing a “United States” location in the document “xmark.xml”.

Thirteen axes are listed in the XML XPath Language specification [14]. The child axis (its mnemonic abbreviation is “/”.) contains the children of the context node. The function getChildNodes in DOM API is used directly for implementing the child axis. The descendant-or-self axis (its mnemonic abbreviation is “//”) contains the context node and the descendants of the context node. The descendant-or-self axis can be achieved by a Depth First Search (DFS) tree traversal. The parent axis contains the parent of the context node (its mnemonic abbreviation is “/../”), if there is one. The function getParentNode in DOM API provides such functionality.
A location step consists of three parts: an axis, a node test and 0 or more predicates. Multiple location steps form a location path which is an XPath expression. After the execution of an XPath query containing a location path, a list of matched nodes is returned. Our query engine supports pipelining of results and a node is being returned at a time by the iterator instead of returning a materialized node list.

3.6 XML Document Loading Engine

Simple API for XML (SAX) is used to parse and load an original XML document. Loading an XML document instance into the disk blocks (pages) means to add the necessary storage-facilitating nodes into the original document and partition the document into small pages. A dynamic bottom-up loading algorithm is introduced in this section. In the following algorithm, MIN stands for the minimum threshold of page fullness, and MAX stands for the maximum threshold of page fullness. If the size of a subtree is in the range between MIN and MAX, a new page is allocated and the subtree is pruned and placed into the new page. Figure 3 shows the process of the algorithm.

Dynamic Bottom-up Loading Algorithm:

1. Perform a depth-first traversal of the document tree.

2. For every node in the node sequence of the depth-first traversal,

3. If the current node is first encountered, mark it as “visited once” and move the current node to the next available node and continue.

4. If the current node is seen again (the “visited once” node), mark it as “visited twice”.
5. If current node is the document root and no more nodes are available, the current subtree is placed into a new allocated page and END (Figure 3e).

6. Calculate the size of the subtree rooted at this current node, denoted as sizeof(current).

7. Get all available previous siblings of the current node, and calculate the sum of the size of subtrees rooted at these siblings, denoted as sizeof(siblings).

8. If MIN <= sizeof(current) + sizeof(siblings) <= MAX, the current subtree and the previous sibling subtrees are pruned and a new c-node is added in the pruned position (Figure 3b, Figure 3d). A new f-node is added as the parent of the current node and the sibling nodes, and the subtree rooted at this f-node is placed in a new allocated page. The current node is set to the new c-node. Mark the new c-node as “visited twice” and go to Step 5.

9. Else If sizeof(current) + sizeof(siblings) > MAX, only the current subtree is replaced with a new c-node even though sizeof(current) could be less than MIN, a new page is allocated (Figure 3c). The current node is set to the new c-node. Mark the new c-node as “visited twice” and go to Step 5.

10. Else If sizeof(current) + sizeof(siblings) < MIN, move the current node to the next available node and continue.

11. End For.
Figure 3a A new page is allocated when sizeof(current) \( \geq \) the minimum threshold of page fullness

Notes: The shaded node indicates that it has been visited once.
The black node indicates it has been visited twice.

Figure 3b A new page is allocated for the current subtree and its sibling subtrees when sizeof(current) + sizeof(siblings) \( \geq \) the minimum threshold of page fullness
Figure 3c A new page is allocated only for the current subtree when sizeof(current) + sizeof(siblings) > the maximum threshold of page fullness.

Figure 3d A new page is allocated for the current subtree and its sibling subtrees when sizeof(current) + sizeof(siblings) >= the minimum threshold of page fullness.
The above algorithm requires that the SAX parser parse the XML document in one pass. The space requirement depends on the height of the document tree. The worst case is that the XML document has a deep recursive structure. For instance, the above algorithm
loads the almost the entire tree into the memory for a "linear" tree. In general, our algorithm successfully loads most XML documents on the “XML Data Repository” website [39].

If the original XML tree structure contains a large fanout, in order to avoid getting a deeply nested page tree, a counter is added to the c-node to store the nested level. Only the c-nodes with the same nested level are allowed to be grouped together and placed in a single page.

4. EXPERIMENTAL RESULTS

In this section, we present the results of experiments that demonstrate the value of our CanStoreX system. We used the XML documents generated by the XMark benchmark [19] with four scale factors, 0.01 (1.1MB), 0.1 (11.3MB), 1 (113.1MB), and 10 (1.11GB). Experiments were run on a 1.54GHz AMD Athlon™ system with 512 RAM running Windows XP professional. All numbers reported are the average of execution times measured in seconds over three executions. The Java Virtual Machine is set to the default allowed maximum memory, which is 64MB. In our experiment, the memory requirement depends on the size of buffer pool. For a buffer pool with a size of 100, the used memory is about 20MB.

4.1 Loading XML documents into CanStoreX

The loading time is linear to the size of the original XML document as shown in Table 2. An approximately 110MB XML document needs 250 seconds, 69,500 disk accesses to be loaded into the CanStoreX system. After the loading is finished, we do a “round-trip” to restore the original XML document. By comparing the tree structures of the restored XML
document with the original one, they are the exactly same, which shows that our loading algorithm is correct. The tree composed of pages has a height of 9 after the 1.11GB XML document is loaded. The small tree height reduces the number of disk accesses when executing a tree traversal.

Table 2 Execution time (seconds) and block accesses when loading XML documents into the CanStoreX system (MRU)

<table>
<thead>
<tr>
<th>factor</th>
<th>file size (MB)</th>
<th>execution time (seconds)</th>
<th>number of block accesses</th>
<th>number of total block requests</th>
<th>hit ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>1.1</td>
<td>3.7</td>
<td>519</td>
<td>1522</td>
<td>65.90</td>
</tr>
<tr>
<td>0.1</td>
<td>11.3</td>
<td>26.7</td>
<td>6818</td>
<td>15230</td>
<td>55.23</td>
</tr>
<tr>
<td>1</td>
<td>113.1</td>
<td>249.2</td>
<td>69498</td>
<td>152134</td>
<td>54.32</td>
</tr>
<tr>
<td>10</td>
<td>1137.4</td>
<td>2466.0</td>
<td>693740</td>
<td>1532270</td>
<td>54.72</td>
</tr>
</tbody>
</table>

4.2 XPath Evaluation

Our experiment uses two queries. The first query (Query 1) is:

/regions/asia/item[location/text()="United States"]

The second query (Query 2) is:

//item[location/text()="United States"]

The first query contains only the location steps composed of child axes. As shown in Table 3, with a limited memory, the CanStoreX system needs about 10 seconds, 1,630 disk accesses to execute the query for an approximately 111MB XML document. This XPath query gets accelerated because the CanStoreX system only needs to traverse the subtree of /regions/asia/ instead of the whole XML document tree. From Table 3, the execution time of Query 1 is linear to the size of the XML documents.
Table 3 Execution time (seconds) and block accesses when executing Query 1 (LRU)

<table>
<thead>
<tr>
<th>factor (MB)</th>
<th>file size (MB)</th>
<th>execution time (seconds)</th>
<th>number of block accesses</th>
<th>number of total block requests</th>
<th>hit ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>1.1</td>
<td>0.67</td>
<td>18</td>
<td>38</td>
<td>52.63</td>
</tr>
<tr>
<td>0.1</td>
<td>11.3</td>
<td>1.95</td>
<td>164</td>
<td>320</td>
<td>48.75</td>
</tr>
<tr>
<td>1</td>
<td>113.1</td>
<td>10.38</td>
<td>1629</td>
<td>3137</td>
<td>48.07</td>
</tr>
<tr>
<td>10</td>
<td>1137.4</td>
<td>89.81</td>
<td>16091</td>
<td>31350</td>
<td>48.67</td>
</tr>
</tbody>
</table>

The second query contains a descendant-or-self axis. It requires a traversal of the whole XML document tree to check all elements. As shown in Table 4, without any indices and with a limited memory, the execution time is about 146 seconds for an approximately 111MB XML document and the number of disk accesses is about 38,035. In order to accelerate such queries an index should be built on our system. From Table 4, the execution time of Query 2 is also linear to the size of the XML documents.

Table 4 Execution time (seconds) and block accesses executing Query 2 (LRU)

<table>
<thead>
<tr>
<th>factor (MB)</th>
<th>file size (MB)</th>
<th>execution time (seconds)</th>
<th>number of block accesses</th>
<th>number of total block requests</th>
<th>hit ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>1.1</td>
<td>2.55</td>
<td>383</td>
<td>529</td>
<td>27.60</td>
</tr>
<tr>
<td>0.1</td>
<td>11.3</td>
<td>16.88</td>
<td>3810</td>
<td>5296</td>
<td>28.06</td>
</tr>
<tr>
<td>1</td>
<td>113.1</td>
<td>146.17</td>
<td>38035</td>
<td>52977</td>
<td>28.20</td>
</tr>
<tr>
<td>10</td>
<td>1137.4</td>
<td>1422.53</td>
<td>383070</td>
<td>533625</td>
<td>28.21</td>
</tr>
</tbody>
</table>

4.3 Comparison of different buffer manager replacement algorithms

In order to study what replacement algorithm is better for the buffer manager in our system. Six replacement policies are implemented and tested. They are: first in first out (FIFO), least recently used (LRU), most recently used (MRU), least frequently used (LFU), most frequently used (MFU), and random replacement (Random). The measurement results (Hit Ratio) are shown in Figure 4.
Figure 4 shows that the MRU has a highest hit ratio when loading XML documents into the CanStoreX system. After a page is generated, especially for the leaf page, it will not very often read it again from the disk. So the most recently used page can be replaced when a loading action is performed. LRU shows a higher hit ratio when queries are executed. But this conclusion would change when the queries containing join operations are executed. In general, MRU is suitable for loading and LRU for executing queries. Other replacement algorithms, such as FIFO and Random, achieve a high hit ratio for both loading and querying.
5. DISCUSSION

1. When a page is expanded to its DOM representation, usually a DOM representation requires approximately 7-10 times more space. In our research, a page may only contain c-nodes for the very large documents. In this case, this page contains a large number of small element nodes with short tag names, which requires more space (20-50 folds) for expansion of its DOM representation in practice. A memory-based DOM implementation under the direct control of our CanStoreX DBMS is necessary.

2. There are some peripheral issues leading to largeness and minor distractions in developing the page-based storage architecture for an XML element. (A) The tag name of an element node can technically be larger than a page. This can be handled by compressing the tag names. We could replace those long tag names with the corresponding shorter ones by a hash function. When retrieving data from the disk, we could restore the hashed tag names from the hash table. (B) An element node can have a very large number of attributes that may not fit on a page. This can be handled by creating another type of storage-facilitating node, called “a-node”, to split all the attributes into multiple pages. (C) A text node occupies space that is larger than a page. In this case chaining additional pages provides a reasonable solution. An application applied to this node would know how to handle a terminal atomic text node such as this. (D) We do not explicitly deal with the issue of prologs, epilogs, comments and processing instructions. In fact, in our implementation we do not deal with items (A) to (D).
3. Currently, our CanStoreX DOM still is read-only. Based on the requirement of the W3C DOM, DOM should be able to remove/insert nodes. An updatable CanStoreX DOM will make our system more promising.

4. Although the LRU shows a better hit ratio than others, the more suitable combined replacement algorithm should exist. Exploring a better buffer replacement algorithm will make our system more efficient.

5. We have assumed that the document being paginated is a legal XML document. The validation schema for a XML document is very important, which will make sure the XML document is legal and error-less. Adding the validation checking functionality is necessary for our system.

6. Indices built on the system can accelerate the running time of the query. Our system is amenable for all kinds of indices.

7. Although XPath is supported in our system, the complicated queried, such as XQuery FLWR expressions, should be able to be evaluated. A suitable evaluation plan and algebra for XQuery needs to be done for our system.

8. The experimental results show that the CanStoreX system can handle XML data that are up to two orders of magnitude larger than what is currently possible. Some hints are predicted in here to make the CanStoreX storage highly scalable to handle terabyte data. We need to test the CanStoreX system for the XML documents with the size of up to 1 terabyte.
REFERENCES CITED


39. XML Data Repository.
ACKNOWLEDGEMENTS

I would like to express sincere gratitude to my major professor, Dr. Shashi K. Gadia, for all of your guidance, support, encouragement and patience. You might not know that your diligence, persistence and deep love for career and life has an invaluable influence on my life.

I would also like to pay my respect and gratitude for my committee members, Dr. Xiaogiu Huang and Dr. Daniel Berleant, for all your time, advice and help.

I would like to thank my fellow students in Computer Science Department for your friendship, enthusiasm and care. Especially, I would like to thank Seo-Young Noh for giving me some very helpful suggestions about my thesis.

I dedicate this work to my parents and family members who made me capable of achieving such heights. I would like to thank my wife, Sa Xu. Without your love and support, I would not be what I am today.