Designing an XML Database Engine: API and Performance

Sudhanshu Sipani, Kunal Verma,
Senthilanand Chandrasekaran, Xiao-Qing Zeng, Jianping Zhu, Dongsheng Che, Kai Wong

Advisor : Dr. John A. Miller
Department of Computer Science
University of Georgia, Athens, GA
jam@cs.uga.edu

Abstract

XML (eXtensible Markup Language) is fast becoming the common electronic data interchange language between applications. In this paper we describe a query processing engine called 'Db4XML'. 'Db4XML' provides storage for XML documents in native format. 'Db4XML is a high performance, main memory resident database engine. The key features of the 'Db4XML' engine is its simplified storage and indexing scheme. This paper explores different query evaluation techniques and discusses concurrency control issues for XML databases. 'Db4XML' supports transactions and provides atomicity at granularity of transactions. We describe the engine's API, architecture, storage, concurrency control and recovery scheme.

1. Introduction

XML is the language of the Web. Several strategies have been proposed to store XML data *viz.* files, relational database, object manager. An XML document can be stored in a relational database by splitting it into number of tables. This leads to inefficiency in querying and storing XML data [11]. Simple queries may require multiple joins. In contrast, a native XML database stores the hierarchical structure of an XML document in native format. This leads to more efficient querying and storing of data.

A database engine is a part of the query processing system of a database. An engine evaluates the optimized query plan. The 'Db4XML' engine is a general-purpose database engine that can work with multiple languages, *e.g.,* XQuery and other alternatives. The key components of a database engine are storage structure, query evaluation system, indices, concurrency control and recovery. The database engine discussed in this paper is part of the 'Db4XML' project underway at the University of Georgia.

A native XML database may contain documents, which conform to a DTD (Document Type Definition). The 'Db4XML' engine uses DTD information to evaluate queries. Indexing scheme for a native XML database relies on the storage structures used. The storage structure used in 'Db4XML' allows us to use simple and efficient indexing schemes for query evaluation. Because of the nature of storage and indexing schemes in 'Db4XML', a path index is almost always used to evaluate queries. Little work has been done on concurrency control for native XML databases [4,5]. 'Db4XML' supports transactions and provides atomicity at the granularity of transactions. This paper discusses different problems encountered while using locks on data with a DOM (Document Object Model) structure. Most of the work on native XML databases has been done on disk-resident databases. 'Db4XML' is a memory resident database, which allows us to use main memory optimization techniques [3].

2. Related work

Several projects have been undertaken for storing and managing the XML data. Lore (Light weight Object REpository), developed at Stanford, is a DBMS for storing and managing semi-structured data [4]. QuiXote [5] is an XML query processing system developed by IBM. QuiXote [5] consists of two parts, the preprocessor and the query processor. The Preprocessor extracts schema for documents and computes structural relationship sets from them, which are used extensively to reduce the query execution time. Xset [6] is a main memory database and search engine, which uses XML as the data storage language. It is a high performance XML database with partial ACID (Atomicity, Consistency, Isolation, Durability) database transcation properties.

Design of a native XML database starts with a data model. Lore [4] uses the OEM model. The OEM (Object Exchange Model) is designed for semi-structured data, which can be seen as a labeled directed graph. The QNX data model used by QuiXote [5] views an XML repository as a set of <schema, setOfData> pairs, where every schema has a set of documents that conform to it. Experiments have been done on storing the XML document as set of tables in relational database [8,9,17]. Lore [10] uses four types of indices *viz.* value index, text index, link index and a path index. The link index provides 'parent pointers' since they are not supported by its object manager. QuiXote [5] uses three kinds of indices for each document, *i.e.,* value index, structure index and link index corresponding to the link relationships.

The 'Db4XML' engine is also a main memory resident database[1][2]. Xset [6], which is a memory resident database, provides atomicity at the granularity of operations and has a simple recovery scheme. Dali [1] discusses a general architecture of a main memory storage manager. System-M is a transaction processing test-bed for evaluating various check-pointing and recovery strategies with different types of logging for immediate update strategy [2].

3. Components

The figure below shows the key components of 'Db4XML' engine. The Query Evaluation system provides an interface to the Query Parser for evaluating an optimized query plan.

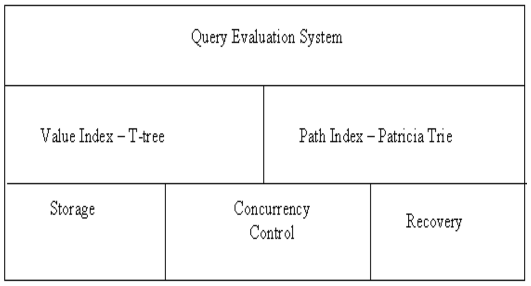

Fig 3.1 Components of 'Db4XML' engine

Query evaluation is supported by two types of index structures. The hierarchical (DOM) structure of the XML documents allows for use of a path index. A value index also supports query evaluation. The engine supports multiple transactions. The engine stores XML data as a hash table of objects keyed by a unique identifier. The engine also stores DTD / schema information. The engine provides concurrency control and uses a recovery strategy to go along with a deferred update strategy to ensure persistence of data. 'Db4XML' has a graphical user interface for performing queries. The GUI provides tools such as adding a schema to the engine's meta-data catalog, adding XML documents to the engine's data repository and performing queries.

4. Storage and Index Structures

XML documents can be stored in the form of relational tables, objects, attributes and B-Tree [8,9]. Results show that the object approach outperforms the relational DTD approach for fixed point queries [8]. Presence of schema and optimizations made on the basis of it can improve performance considerably [5,9]. 'Db4XML' stores DTD / schema [section 4.2] and uses type information generated in the storage of DTD / schema in storing XML document [section 4.3] for query evaluation.

4.1 Data Model

An XML document can be viewed as a directed graph. A DOM (Document Object Model) for an XML document without references is a tree structure, which details the hierarchical information and parent-child relationship between various elements of the document. If links are added to an XML document the document can be modeled as a graph.

A DTD (Document Type Definition) for XML schema provides information about various types of elements contained in an XML document. Consider the 'employees' DTD (employees.dtd) which we will use as an example in the rest of this paper.

```
<!ELEMENT employees (employee+)>
<!ELEMENT employee (name, salary, department)>
<!ATTLIST employee id CDATA #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT salary (#PCDATA)>
<!ATTLIST salary payperiod CDATA #IMPLIED>
<!ELEMENT department (title+)>
<!ELEMENT title (#PCDATA)>
```

Consider an 'employees' XML document which conforms to the DTD mentioned above

```
<!DOCTYPE employees SYSTEM "employees.dtd">
<employees>
    <employee id="001">
         <name>Greg</name>
         <salary payperiod="weekly">1500</salary>
         <department>
                      <title>Developer</title>
         </department>
    </employee>
    <employee id="002">
         <name>Mark</name>
          <salary payperiod="weekly">2000</salary>
          <department>
                    <title>Manager</title>
          </department>
    </employee>
    <employee id="003">
         <name>John</name>
         <salary payperiod="weekly">2000</salary>
          <department>
                   <title>Manager</title>
          </department>
    </employee>
</employees>
```

The document mentioned above can be represented as a graph shown in Fig 5.1

4.2 Meta-Data

A DTD for XML schema in 'Db4XML' system is stored in the form of *ElementType*s, which is at the lowest level of granularity for storing meta-data for XML documents. Each *ElementType* has a unique identifier (*elementTypeId)*. To account for hierarchical storage of data in a DTD, each *ElementType* has an identifier of its parent denoted as *parentTypeId* and a collection of identifiers of all its children.

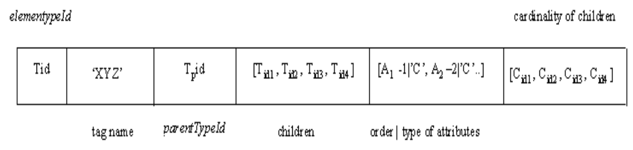
elementypeId
cardinality of children
Tid
'XYZ'
$T_p$id
$[T_{id1}, T_{id2}, T_{id3}, T_{id4}]$
$[A_1$ -1|'C', A_2 –2|'C'..]
$[C_{id1}, C_{id2}, C_{id3}, C_{id4}]$
tag name
parentTypeId
children
order | type of attributes

Fig 4.2.1 *ElementType* is at the lowest level of granularity for 'Db4XML' meta-data

The *Children* collection is an ordered list and hence this scheme supports order queries. *ElementType* stores attribute information in a hash map, which maps attribute name to its order (the position of attribute in its DTD definition for an *ElementType*). All *ElementType*s for a DTD are stored in a hash map, which maps each *elementTypeId* (identifier) to its type information (meta-data). *ElementType* also stores the tag name of the DTD node and cardinality of its children. The cardinality of each child (represented using ? / * / +) is stored in an array as show in Fig 4.2.2.

The XML 1.0 specification allows element attributes to have reference types (IDREF, IDREFS, *etc.*) [12]. This information is also stored in the hash table, which maps the attribute name to the order of the attribute using separators. The 'employees' DTD shown previously can be represented as in the following map

tag name	*elementTypeId*	*parentTypeId*	children	order / type of Attributes	cardinality
employees	1	-1	[2]	[]	['*']
employee	2	1	[3,4,5]	[id - 1|'ID']	[1,1,1]
name	3	2	[]	[]	[]
salary	4	2	[]	[payperiod - 1|'C']	[]
department	5	2	[6]	[]	['*']
title	6	5	[]	[]	[]

Fig 4.2.2 'employees' DTD stored as a table

4.3 XML Document Storage in Native Format

The data in the 'Db4XML' system are stored in form of *Element*s, which is the lowest level of granularity for the XML data. Each Element node in the DOM tree maps to an *Element* object in Db4XML. Each element is identified by a unique identifier (*elementId).* To account for hierarchical storage of data in any native XML database, each element has a *parentId* and a collection of *elementId's* of all its children.

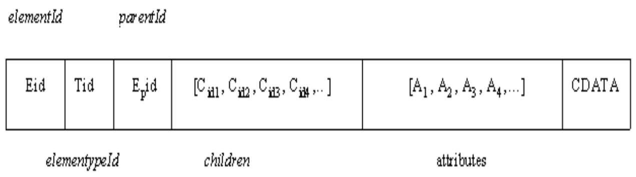
elementId
parentId
Eid
Tid
$E_p$id
$[C_{i1}, C_{i2}, C_{i3}, C_{i4}, ..]$
$[A_1, A_2, A_3, A_4, ...]$
CDATA
elementypeId
children
attributes

Fig 4.3.1 *Element* is at the lowest level of granularity for 'Db4XML' storage

Each element is associated with a type identifier. A type identifier is a unique identifier for type associated with that particular element, which is derived from the DTD, to which the document conforms. For this reason 'Db4XML' does not store tag names with the data, which leads to considerable saving in terms of space. Each element has an ordered list of attribute values and CDATA value. The list of attributes can also contain inter document references as per the XML 1.0 specification. Data in 'Db4XML' is a hash table, which maps each element identifier (*elementId*) to an element. The 'employees' XML document can be represented as below.

elementId	*typeName*	*elementTypeId*	*parentId*	*children*	Attributes	CDATA
1	employees	1	-1	[2,7,12]	▯	
2	employee	2	1	[3,4,5]	['001']	
3	name	3	2	▯	▯	Greg'
4	salary	4	2	▯	['weekly']	1500
5	department	5	2	[6]	▯	
6	title	6	5	▯	▯	Developer'
7	employee	1	1	[8,9,10]	['002']	
8	name	2	7	▯	▯	Mark'
9	salary	3	7	▯	['weekly']	2000
10	department	4	7	[11]	▯	
11	title	5	10	▯	▯	Manager'
12	employee	6	1	[13,14,15]	['003']	
13	name	1	12	▯	▯	John'
14	salary	2	12	▯	['weekly']	2000
15	department	3	12	[16]	▯	
16	title	4	15	▯	▯	Manager'

Fig 4.3.2 'employees' XML document stored in a native format

4.4 Index structures in 'Db4XML'

Relational databases are traditionally queried with associative queries, retrieving tuples based on values of some attributes. In object-oriented databases, the concept of Path index is specified to relate attributes to objects. A Path index is an index on the path expressions (*e.g.* /employees/employee) of the XML document. Path index is used in Db4XML because of the nature of XML data. Several data structures can be used to answer path queries in XML, *e.g*., Hash table, Trie, *etc*. 'Db4XML' uses a Patricia trie[9]. The Patricia trie helps in efficient evaluation of queries with generalized path expression.A Patricia Trie node can store list of the elements, according to their particular path. Alternatively a hash table could be used to implement a path index, but our tests have shown that it is inefficient to evaluate queries with generalized path expressions using just a hash index.

5. Query Evaluation

The 'Db4XML' engine provides an API for evaluating queries. The engine uses path and value indices and can evaluate queries using different techniques.

Consider the XQuery
```
FOR $e IN /employees/employee
WHERE $e/salary < 2000
RETURN
        <employee eid = {$e/@id}>
                {<name> { $e/name } </name>}
                {<salary>{$e/salary } </salary>}
        </employee>
```

The engine can take one of the following approaches towards evaluating the above query :

1. A recursive approach can be used to evaluate a query. In this approach tag names of the elements can be matched with the path expression in the query at each successive step. This can be inefficient for larger documents.
2. Another approach could be to get a list of all the elements having the path "/employees/employee" using the path index(Fig 5.1). The conditions are evaluated by recursively going down the DOM tree for the XML document, starting from the "/employees/employee" node and finding the "/employees/employee/salary" nodes and checking whether

the conditions are satisfied. Alternatively, "/employees/employee/salary" elements can be located first using the path index. Then parent pointers can be followed to get their parent information.

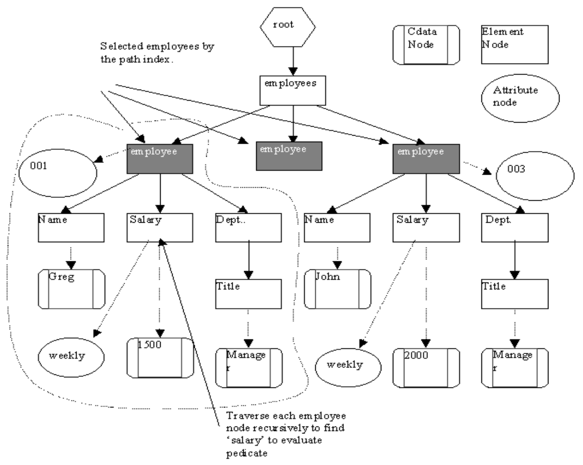

Fig 5.1 Using Path index to evaluate query.

3. The approach mentioned above can still be costly if we have an XML document with 50,000 employee elements. In this case, a Value index can be used. The value index can return a list of elements with their CDATA / attribute values as given in the predicate (2000 in the example query above). We can traverse up the tree from the "/employees/employee/salary" element, until we find out to which "/employees/employee" element it belongs.

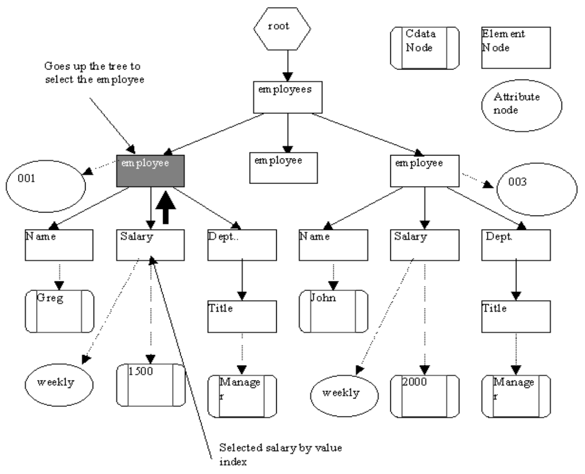

Fig 5.2 Using Value index to evaluate query

6 Engine Architecture and Implementation

In this section, we discuss the engine's API. 'Db4XML' is a memory resident database. Fig 6.1 shows the various data structures in the main memory at run time. This includes XML data, meta-data, indices (path and value), information about acquired locks (`LockTable`) and information about active transactions (Active `Transaction Table`).

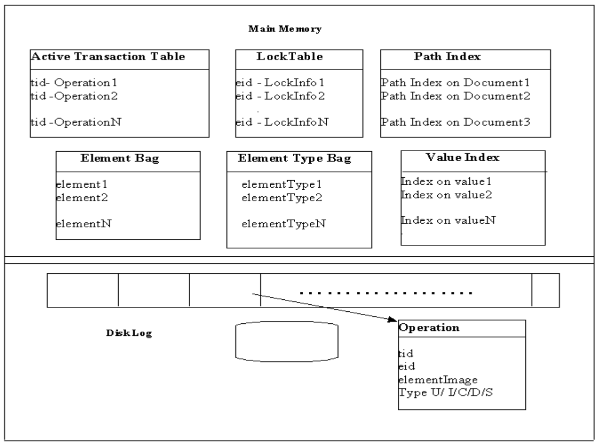

Fig 6.1 Data structures at runtime.

6.1 UML Design

The main class in our system is the `ElementBag`, which provides a remote API for the Query Processing module. `ElementBag` has references to all other objects in the system and depending on the remote method invocation, the appropriate method of the appropriate object is called. It also has a hash table, which contains all the elements in the database. An `Element` is a representation of an element node in an XML document and the meta-data for it is defined in a corresponding `ElementType`. `ElementType` objects are stored in a hash table called `ElementTypeBag`.

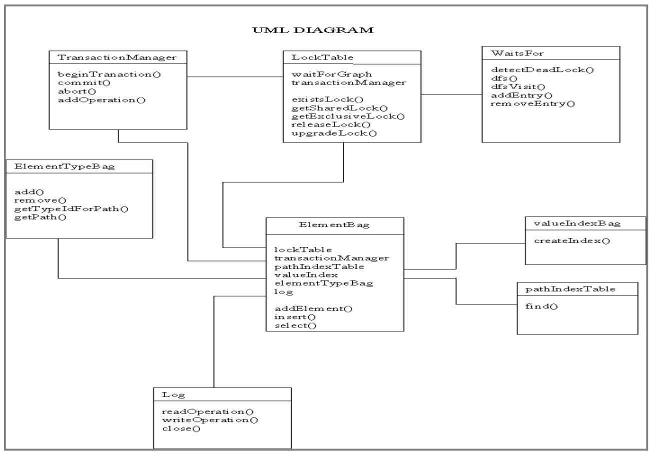

Fig 6.1.2 UML diagram of all the classes.

In order to control concurrency, there is a `TransactionManager` class that maintains the identifiers of all transactions and a list of all the operations performed by them. Along with that there are two other classes that maintain locking information - `LockTable` and `LockInfo`. Every time a transaction waits, the waitsfor graph is updated in `WaitsFor` to check for deadlock.

There are two main classes for indexing, `ValueIndex` and `PathIndexTable`.

6.2 Sequence of Events

Here is a description of the sequence of events for a *select* operation . Since the remote interface is

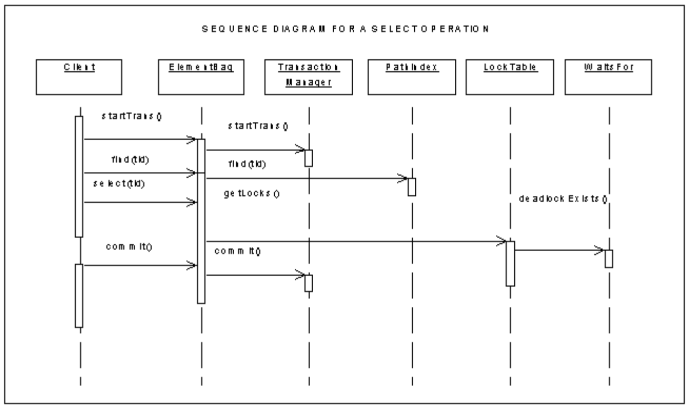

Fig 6.2.1 Sequence diagram.

provided by `ElementBag`, the processing module, which has a remote reference to the `ElementBag` object calls the `ElementBag`'s `startTransaction`() method to get a transaction identifier for that transaction. All subsequent method invocations by the transaction must have that transaction identifier in them.

The `find`() method is then invoked on `PathIndexTable`, which returns element identifiers (*elementIds*) of all the `Elements` having a particular path. Then an appropriate `select`() method is called and during the execution of `select` shared locks are requested from the LockTable for all elements being read. Either a lock is granted or the transaction waits to acquire that lock. If a transaction waits, the `WaitsFor` graph is updated and a depth first search is performed to check for deadlocks. In case of a deadlock the transaction will be aborted. In order to improve concurrency, read locks are requested on all elements being read, however, the locks are only kept if the element satisfies all the select conditions (conjunctive). After that all the selected element identifiers (*elementIds*) are returned to the query processor. The query processor can invoke `commit`() to end the transaction. The `commit`() method of `ElementBag` in turn calls the `commit`() method of the `TransactionManager`.

7. Transaction Management in Db4XML

7.1 Concurrency Control

In relational databases all tuples being read or written by a transaction are locked along with index structure. The hierarchical structure of DOM and the parent-child relationships in the structure of the XML document presents a different problem.

Consider the following Xquery command,

```
FOR $e IN /employees/employee
WHERE $e/salary < 2000 and  $e/@id < 005
RETURN
        <employee >
        {<name> { $e/name } </name>}
        </employee>
```

This query checks for elements having path '/employees/employee/salary' and id of the 'employees/employee' element and returns '/employees/employee/name' element. Suppose a transaction T_1 runs the query on the engine and obtains write locks on all '/employees/employee/name' elements selected. At this point, any transaction T_2 can do any of the operations mentioned below and commit before the transaction T_1 commits. This can violate the ACID properties of transactions. 1) T_2 can delete the '/employees/employee' element to which '/employees/employee /name' element belongs (ancestor). 2) T_2 can alter the id attribute of the '/employees/employee' element to which the '/employees/employee/name' element belongs. 3) T_2 can alter the CDATA of the '/employees/employee/salary' element in hierarchy which is associated with the 'employees/employee/name' element.

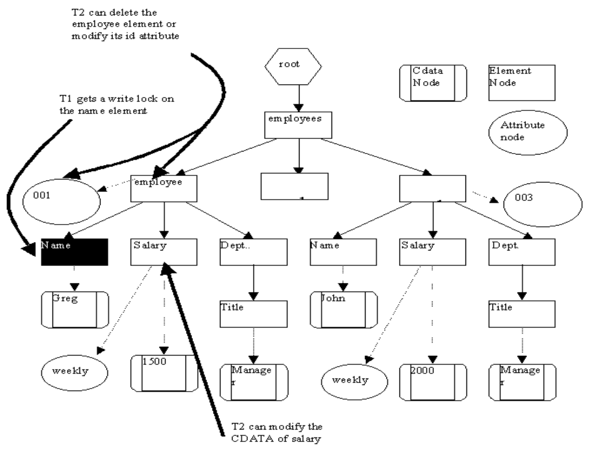

Figure 7.1.1 Problems while locking DOM tree.

Following approaches can be used for locking elements of XML documents:

1. Whenever a transaction reads elements in a top down fashion, it acquires read locks for element nodes being traversed (top down approach)
2. All elements read/written are locked and the locks are released for those elements which do not satisfy the predicate. Locks on all the elements which are read and satisfy the WHERE clause are retained. As in the example above, locks on element 'employees/employee/salary' and 'employees/employee/@id' will be maintained, for all the

elements in the employee substructure which satisfy both conditions.

3. If a transaction deletes an element node, it has to acquire write locks on all the descendants of that element node. This gains significance when a value index or path index is used to answer queries.

7.2 Implementation Overview

A transaction is a sequence of operations. A transaction gets a unique identifier from the engine and all the further operations in the transaction invoke methods on the engine using that unique identifier (*transactionId*). A transaction has to obtain read/write locks on indices, elements and ancestors, which are being read or written. If all the operations are successful, the transaction commits. We have used rigourous-2PL as the locking scheme, as a result of which a transaction releases all its locks only during its commit operation. The engine uses a deferred update scheme, as a result of which elements are not updated until commit point. Instead an image of the updated element, along with the type of operation is stored in an ATT (Active Transaction Table) for every operation. The engine also maintains a graph of waiting transaction and uses a depth first search algorithm for deadlock detection. During commit all the operations in the ATT for the transaction are written to the disk resident log, all the locks are released and all the instances of the transaction from deadlock detection graph are removed. When a transaction is aborted all the locks are released, all the instances of the transaction are removed from the deadlock detection graph and the operation history of the transaction from the ATT is removed.

7.3 Recovery

'Db4XML' engine uses a deferred update strategy. Using the deferred update strategy simplifies the recovery scheme and procedure. The 'Db4XML' engine uses a log on disk, which contains a sequence of operations for all the committed transactions. The log is the database. If operations are being added to the log, the log could be infinitely growing. Hence, only the most recent updates or inserts (on elements) are recorded in the log .

During the recovery process the log can be read sequentially and all the operations redone. To account for failure during commit, the log need to be scanned once to find the list of committed transactions. The log then can be read serially and all the operations can be redone only for the committed transactions. This method is under implementation.

8 Conclusions and Future Work

In this paper, we have described the key components of the 'Db4XML' database engine and their functionality. We have used an efficient data model for storage of XML documents. The data model allows simpler and fewer index structures. Our storage and indexing structure is less complicated than that of QuiXote and Lore. While Lore uses value indices, link indices, text indices and path indices, we have used path indices and value indices. Empirical studies on using different type of indices and different query evaluation strategies are currently underway. We also need to provide text indices. Little work has been done on concurrency control issues for XML databases. While the techniques used are variations of the traditional concurrency control and recovery methods, applying them to an XML storage model (DOM) requires added care. We have also implemented a recovery scheme based on deferred update strategy. In the future we plan to incorporate other techniques for checkpointing, concurrency control and recovery.

References

[1] P. Bohannon, R. Rastogi, D. Lieuwen, S. Seshadri, A. Silberschatz and S. Sudarshan. The architecture of the Dali main-memory storage manager. *In Journal of Multimedia Tools* and Applications, 1997.

[2] H. Garcia-Molina, K. Salem: System M: A Transaction Processing Testbed for Memory Resident Data. *In IEEE Transactions on Knowledge and Data Engineering*, March 1990, Vol. 2, number 1161#172.

[3] H. Garcia-Mollina and K. Salem, Main memory database system: An overview. *In IEEE Transactions on Knowledge and Data Engineering*, Volume 4, Number 6, 1992.

[4] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *In SIGMOD Record*, September 1997.

[5] M . Mani and N. Sundaresan , Query Processing using QuiXote. IBM research report.

[6] Ben. Y. Zhao, Anthony.D.Joseph :Xset: A lightweight database for Internet Applications. Master's thesis. Computer Science Division, UCLA, Berkeley.

[7] R.Malhotra, XML database engine. Master's thesis, University of Georgia. http://chief.cs.uga.edu/~jam/home/theses/rakesh_thesis/rakeshthesis.ps

[8] Feng Tian, David J. DeWitt, Jianjun Chen, and Chun Zhang. *Submitted*. The Design and performance evaluation of alternative XML storage strategies. http://www.cs.wisc.edu/niagara/papers/xmlstore.pdf

[9] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason and Moshe Shadmon. *A fast index for semistructured data*. *In VLDB*, 2001

[10] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing Semistructured Data. Technical Report, January 1998

[11] Relational Databases for Querying XML Documents: Limitations and Opportunities. Jayavel Shanmugasundaram,

H. Gang, Kristin Tufte, Chun Zhang, David DeWitt, and Jeffrey F. Naughton. *In VLDB,* 1999.
[12] eXtensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 6 October 2000, http://www.w3.org/TR/REC-xml