Designing a High Performance Database Engine for the 'Db4XML' Native XML Database System

Sudhanshu Sipani[a], Kunal Verma[a], John A. Miller[a, *] and Boanerges Aleman-Meza[a]

[a] *Department of Computer Science, University of Georgia, Athens, GA 30602*

Abstract

XML (eXtensible Markup Language) is fast becoming the common electronic data interchange language between applications. In this paper, we describe a database engine called 'Db4XML', which provides storage for XML documents in native format. 'Db4XML' is a high performance, main memory resident database engine. 'Db4XML' is being used as a testbed for comparing various query evaluation techniques. The use of wild card (*, ?, *etc.*) in the path expressions of a query allows users to query documents whose structural information is not available. This paper lists different techniques that can be used to evaluate Generalized Path Expressions (GPE) and presents a performance comparison of the same. A preliminary performance study of the effect of using concurrency control techniques on the various query evaluation techniques is also performed. This paper briefly discusses a suitable recovery technique for the database engine.

Keywords: native XML database, database engine, transaction management, concurrency control, performance evaluation

[*] corresponding author. *Address:* Department of Computer Science, University of Georgia, Athens, GA 30602. *E-mail* – jam@cs.uga.edu. *Telphone:* 706-542-3440. *Fax:* 706-542-2966

1 Introduction

XML is being increasingly used as data interchange language. This has created opportunities for storing and managing XML data. Much work has been done on various methods for storing and querying XML data [McHugh et al., 1997; Mani and Sundaresan, 1996]. A number of commercial database systems have sprung up that exclusively store and manage XML data. Commercial XML databases are either Native XML Databases or XML–Enabled Databases. A Native XML Database is a database, which is specifically designed for storing and querying XML documents. It contains data structures to maintain the hierarchical structure of XML data and uses its knowledge about them to optimize query processing. An XML–Enabled database typically uses additional layers to map XML data to its underlying storage structures. There is little known performance results in regard to Native XML Databases. Native XML Databases have gained popularity from eBusiness technologies, where XML is the *de facto* standard for B2B information exchange. Leading vendors in the XML Database market include Software AG with their product Tamino (The Tamino XML Server, 2002), and "eXtensible Information Server" from eXcelon Corp (eXtensible Information Server (XIS), 2002).

The 'Db4XML' native XML database has the following components – Query Tool, Query Processor and the 'Db4XML' Database Engine. The Query Tool has a Graphical User Interface (GUI) for user interaction. The GUI provides tools such as adding a schema to the meta-data catalog, adding XML documents to the data repository and performing queries. A GUI tool for querying XML documents is discussed in (Miller and Sheth, 2000). The Query Processor parses the query and generates an optimized query evaluation plan for the engine. This paper

provides a description of the implementation and the performance of the 'Db4XML' database engine.

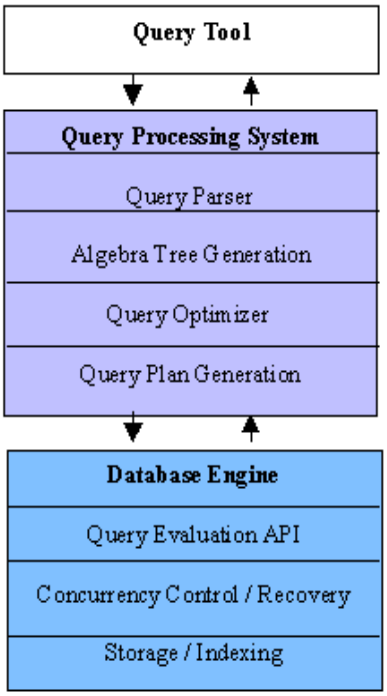


Figure 1. Components of the 'Db4XML' system

A database engine is the component of a database, which maintains storage and indexing structures, facilitates transaction management and provides security. It is the only component that can directly access data and it provides an interface for the other components to manipulate the data. An engine evaluates the optimized query plan. The 'Db4XML' engine is a general-purpose database engine that can work with multiple query languages, *e.g.,* XQuery (XQuery 1.0, 2001) and other alternatives. XQuery is the W3C recommended language for querying XML data. Though the engine supports the XQuery language, it is sufficiently low-level to support other query languages. The key components of the 'Db4XML' database engine are

storage structures, query evaluation API, indices, concurrency control and recovery. The 'Db4XML' engine has its own native representation of the XML data and a storage structure that allows it to maintain both hierarchical and non-hierarchical relationships between the various elements in an XML document. A native XML database may contain documents, which conform to a DTD (Document Type Definition) or XSD (XML Schema Definition). The 'Db4XML' engine stores DTD / XSD information and uses this information to evaluate queries. The engine provides a rich API to evaluate queries using multiple techniques. The engine is used as a testbed to compare different query evaluation techniques. The indexing schemes for a native XML database rely on the storage structures used. The storage structure used in 'Db4XML' allows us to use simple and efficient indexing schemes for query evaluation.

A DOM (Document Object Model, 2002) for an XML document without links is a tree structure, which details the hierarchical information and the parent-child relationships between various elements of the document. The engine provides concurrency control techniques that solve different problems encountered while using locks on data with a DOM structure. The engine uses a recovery strategy to go along with a deferred update strategy to ensure persistence of data. Most of the work on native XML databases has been done on disk-resident databases. 'Db4XML' is a main memory resident database, which allows us to use main memory optimization techniques (Garcia-Molina and Salem, 1992). The database engine discussed in this paper is a part of the 'Db4XML' project underway at the University of Georgia. Overall design issues for the database engine are introduced in (Sipani et al., 2002).

This paper is organized as follows. Section 2 gives an overview of related work in this area. Sections 3 and 4 discuss the storage and indexing structures used by the 'Db4XML' engine. The architecture and implementation of the 'Db4XML' engine is discussed in section 5. Section 6 presents the query evaluation techniques for querying XML data and a performance study of

the various techniques. Section 7 gives an overview of transaction management in the 'Db4XML' engine and the issues involved in controlling access to the XML data. The section also studies the effect of concurrency control on the various query evaluation techniques. Finally, conclusion and future work are mentioned in section 8.

2 Related Work

Several projects have been undertaken for storing and managing XML data. Lore (Light weight Object REpository), developed at Stanford, is a DBMS for storing and managing semistructured data (McHugh et al. 1997). QuiXote (Mani and Sundaresan, 1996) is an XML query processing system developed by IBM. The QuiXote system consists of two parts, the preprocessor and the query processor. The preprocessor extracts schema information from documents and computes structural relationship sets from them, which are used extensively to reduce the query execution time. XSet (Zhao and Joseph, 1999) is a main memory hierarchically structured database with partial ACID properties. XSet (Zhao and Joseph, 1999) does not support transactions and provides atomicity at the level of individual operation only. Niagara (Naughton et al., 2000) is an Internet query system with an XML query processing system that is suitable for querying the Internet and provides mechanism to 1) find the XML files that are relevant to the query and 2) deal with remote data sources efficiently.

Design of a native XML database starts with a data model. Lore (McHugh et al. 1997) uses the OEM model. The OEM (Object Exchange Model) is designed for semistructured data, which can be seen as a labeled directed graph. The QNX data model used by Quixote (Mani and Sundaresan, 1996) 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 [Shanmugasundaram et al., 1999]. Lore uses five types of indices *viz.* value index, text index, link index, text index and a path index. The

link index provides 'parent pointers' since they are not supported by its object manager. Quixote (Mani and Sundaresan, 1996) uses three kinds of indices for each document, *i.e.*, value index, structure index and link index corresponding to the link relationships. Niagara (Naughton et al., 2000) caches the XML documents, which have been searched previously. Its search engine maintains three types of indices: element lists, word lists and DTD lists. The element list associated with a given element name stores information about the files that contain the XML element, and the position of the XML element in those files. Similar types of lists are maintained for each word and DTD encountered in the searched XML files.

An important factor in the design of databases is the storage scheme. There is a good contrast between disk resident and main memory resident databases (Garcia-Molina and Salem, 1992). Dali (Bohannon et al., 1997) discusses the general architecture of a main memory storage manager. XSet (Zhao and Joseph, 1999), which is a main memory resident database, provides atomicity at the granularity of operations and has a simple recovery scheme. System-M is a transaction processing testbed for evaluating various checkpointing and recovery strategies with different types of logging for immediate update strategy (Garcia-Molina and Salem, 1990).

3 Storage Structures in 'Db4XML'

The following approaches have been used for storing XML documents: text files with indices, object oriented databases, relational databases and native XML databases. While using text files with indices, the whole index needs to be updated for a small change in the XML document. Various schemes have been suggested for mapping XML data into relational databases (Florescu and Kossman, 1999; Shanmugasundaram et al., 1999; Arpinar et al., 2001). Performance evaluation of various XML storage strategies (Tian et al., 2001) suggests that information from XSDs or DTDs are essential for good performance. The presence of schema

and optimizations made on the basis of it can improve performance considerably (Tian et al., 2001). The 'Db4XML' engine stores information from DTD / XSD as meta-data and uses it in storing XML documents and for query evaluation.

3.1 Data Model

An XML document can be viewed as a directed graph. A DOM (Document Object Model) (Document Object Model, 2002) for an XML document without links is a tree structure, which details the hierarchical information and the parent-child relationships 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) or XSD (XML Schema Definition) defines the document structure and possible elements. It provides information about the various types of elements in an XML document. Here is an example document 'employees' which we will use in

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE employees SYSTEM "employees.dtd">
<employees>

    <employee id="777-97-0936" supervisor="003">
      <name>Greg</name>
      <salary payperiod="yearly">40000</salary>
      <department>
          <title>Marketing</title>
      </department>
    </employee>

    <employee id="999-22-3432">
      <name>Mark</name>
      <salary payperiod="yearly">70000</salary>
      <department>
          <title>Sales</title>
      </department>
    </employee>

    <employee id="888-33-5655">
        <name>John</name>
        <salary payperiod="yearly">60000</salary>
        <department>
            <title>Marketing</title>
        </department>
    </employee>

</employees>
```

Figure 2. An example document ‘employees’

the rest of the paper:

A DTD for the ‘employees’ XML document is shown in Figure 3.

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

Figure 3. A DTD for ‘employees’ document

An XML Schema (XSD) for the example document ‘employees’ is given in Appendix A. The document mentioned above can be represented as a DOM-like graph as shown in Figure 4.

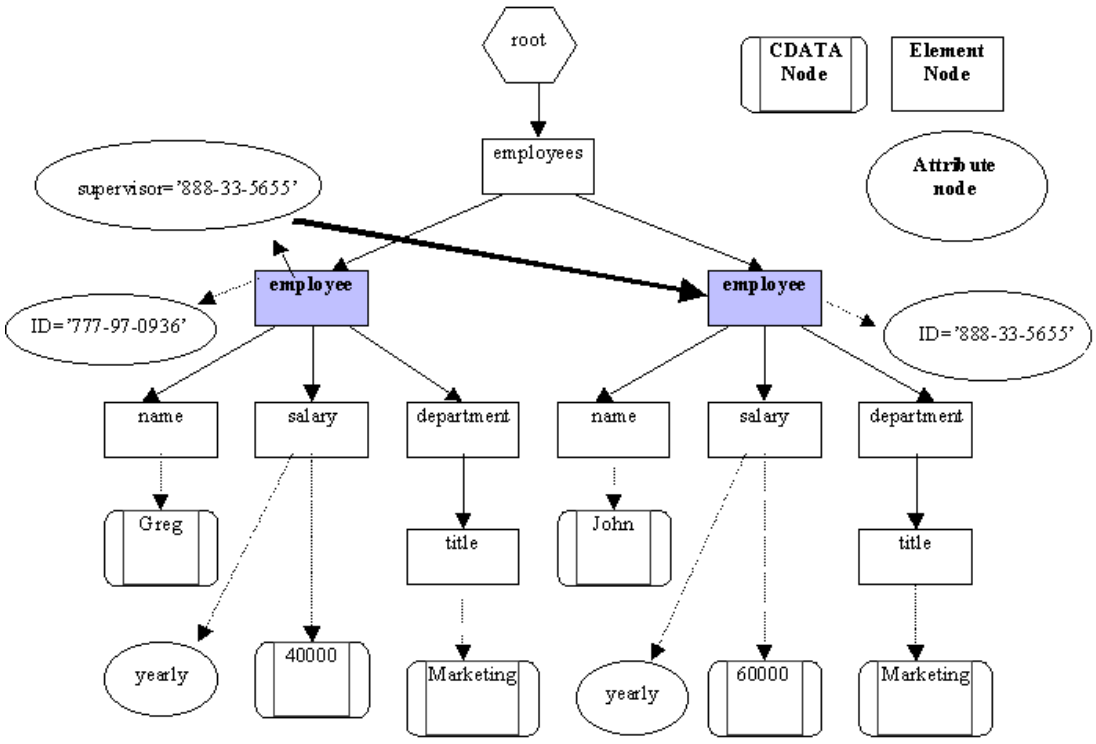


Figure 4. 'employees' document represented as a graph

3.2 XML Document Storage in Native Format

An XML document is stored in the form of `Element` objects. The information about each element node in the DOM tree of an XML document is stored in an object of type `Element`. Each `Element` object is identified by a unique identifier (`elementId`). To account for the hierarchical storage of data, each `Element` stores the `elementId` of its parent node as `parentId` and `elementId`s of all its children.

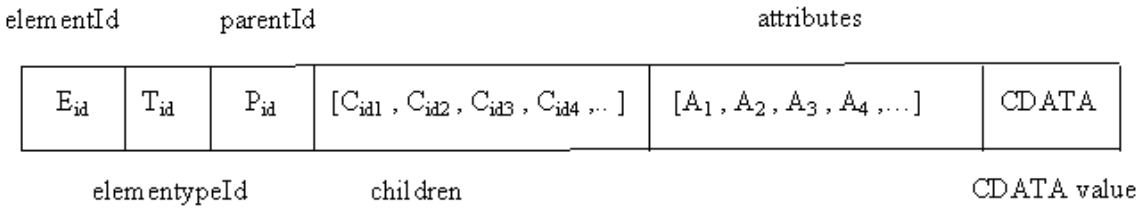


Figure 5. `Element` object for storage of XML documents

Each element has a type identifier `elementTypeId`, which is a unique identifier for the type associated with that particular element. The type information is derived from the DTD / XSD, to which the document conforms. For this reason, 'Db4XML' does not store tag names with the data itself, which may lead to considerable saving in terms of space. Each element has an ordered list of attribute values and a CDATA value. The list of attributes can also contain intra document references as per the XML 1.0 specification (IDREF, IDREFS). Data in 'Db4XML' is stored in a hash table, which maps each element identifier (`elementId`) to an `Element` object. The 'employees' document can be represented as shown in Table 1.

elementId	elementTypeId	parentId	Children	Attributes	CDATA
1	1	-1	[2,7,12]	[]	
2	2	1	[3,4,5]	['777-97-0936' ,'888-33-5655']	
3	3	2	[]	[]	Greg
4	4	2	[]	['yearly']	40000
5	5	2	[6]	[]	
6	6	5	[]	[]	Marketing
7	2	1	[8,9,10]	['999-22-3432','']	
8	3	7	[]	[]	Mark
9	4	7	[]	['yearly']	70000
10	5	7	[11]	[]	
11	6	10	[]	[]	Sales
12	2	1	[13,14,15]	['888-33-5655','']	
13	3	12	[]	[]	John
14	4	12	[]	['yearly']	60000
15	5	12	[16]	[]	
16	6	15	[]	[]	Marketing

Table 1. 'employees' document stored in a native format

3.3 Meta-Data

Meta-data for a database is the information about the data stored in a database *viz*. type, relationships information. For an XML database, it also includes the structural information about the data. Meta-data for XML documents is defined in DTDs / XSDs. A DTD / XSD is parsed and stored in the form of `ElementType` objects. `ElementType` object stores structural information (children, attributes, *etc*.) about an element in a DTD / XSD. Each `ElementType` object is identified by a unique identifier (`elementTypeId`*)*. Each `ElementType` object

stores the `elementTypeId` of its parent as `parentTypeId` and a collection of `elementTypeId`s of all its children. The `children` collection is an ordered list and hence this storage scheme supports ordered queries. The cardinality of each child is also stored. An `ElementType` object stores information about attributes in a hash table, which maps attribute name to its order, reference type (*refType*) and data type (*dataType*). A *refType* can be ID, IDREF or IDREFs (eXtensible Markup Language (XML) 1.0 (Second Edition)., 2001).

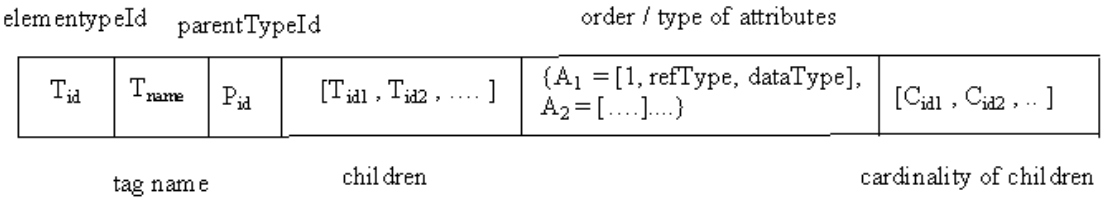


Figure 6. `ElementType` object for storing meta-data information extracted from a DTD / XSD

The information generated from the 'employees' DTD shown in Figure 3 can be represented as shown in Table 2.

elementTypeId	tagName	parentTypeId	children	Order / type of Attributes	Cardinality
1	Employees	-1	[2]	[ ]	['*']
2	Employee	1	[3,4,5]	{ id = [1, 'ID', 'INTEGER'], supervisor = [2, 'IDREF', 'INTEGER']}	['1','1','1']
3	Name	2	[ ]	[ ]	[ ]
4	Salary	2	[ ]	{[payperiod = [1, '', 'STRING']}	[ ]
5	Department	2	[6]	[ ]	['1']
6	title	5	[ ]	[ ]	[ ]

Table 2. Information extracted from the 'employees' DTD

4 Index Structures in 'Db4XML'

A native XML database parses the XML data and stores it into a tree like structure similar to DOM. A query can be evaluated by traversing the DOM structure of an XML document top-down from the root to each node below it until useful information is found. This is,

however, an inefficient approach. That is why index structures are utilized to speed up query execution.

Let us summarize the index structures used in the existing Native XML Database systems. Lore (McHugh et al., 1998) provides a whole array of indices (value, link, edge, text and path). A value index stores references to all the atomic objects (integer, real, string) that have an incoming edge with the same label / tag name. Lore does not have parent pointers. It has a link index operator, Lindex (x,*l*,y), which places into x all objects that are parents of y via an edge labeled *l*. Lore also has an edge index, which holds all the parent child pairs that are connected via a specific label. Lore has a text index, which can be used to search for specific words or group of words. Finally there is a path index. All objects reachable via a specific path can be accessed using the path index. QuiXote (Mani and Sundaresan, 1996) has a text index, index on attribute values and a structure index (path index). XSet (Zhao and Joseph, 1999) has a simple, hierarchical index structure. It is a dynamic structural summary of all the documents in the database. It is like a combination of a path and a value index. The Niagara (Naughton et al., 2000) system caches XML files and creates indices on those files. Niagara uses variants of inverted lists (Knuth, 1973) for indices. Niagara search engine maintains three types of inverted lists: element lists, word lists and DTD lists. The element list associated with a given element name stores information about the files that contain the XML element, and the position of the XML element in those files. Each entry of the element list is of the form (*fileId*, *beginId*, *endId*), where *fileId* identifies the file containing the XML element, *beginId* and *endId* specify the begin and the end position of the element in that file. Similar type of inverted lists are maintained for each unique word and DTD encountered in the searched XML files.

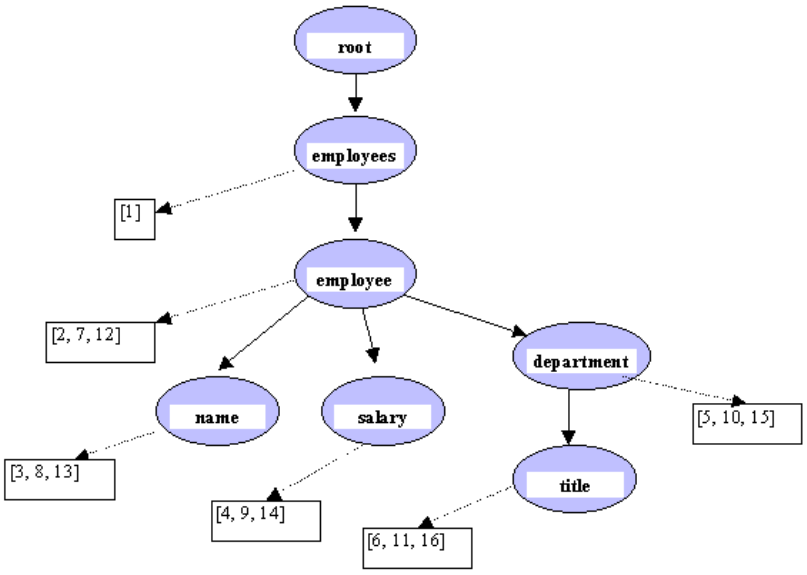
The 'Db4XML' engine currently uses path index and value index for faster query evaluation. Since 'Db4XML' primarily supports those documents whose DTD is known in

advance, it has a path index, which can list identifiers of all the elements, which are reachable via a path. Since elements in our storage have parent pointers, no link index is needed as in Lore. At present, the 'Db4XML' engine does not provide text indices.

4.1 Path Index

Relational databases are traditionally queried with associative queries, retrieving tuples based on values of some attributes. In object-oriented database, the concept of path index is specified to relate attributes to objects. A path is a sequence of names of element nodes in the data model that would be traversed to reach a particular element or attribute node. A path index is an index on the path expressions (*e.g.*, /employees/employee) contained in all the XML documents in a database. It lists references to all the elements reachable via a path expression.

Several data structures can be used to answer path queries in XML, *e.g.*, Hash table, Trie, *etc*. 'Db4XML' uses a Patricia Trie (Cooper et al., 2001). Each node in the Patricia Trie, has a collection of references to element nodes, which are reachable via a path expression. For



example,

Figure 7. A path index for the 'employees' document

to find out the list of elements reachable via a path, *e.g.*, employees/employee/name, the path index shown in Figure 7 can be traversed from the root to the bottom until the element list is found (3,8,13). The Patricia Trie especially helps in the evaluation of queries containing Generalized Path Expressions (GPEs). For example, the GPE employees.*.name can be expanded to employees.employee.name using the path index shown in Figure 7.

Alternate techniques can be used for the construction of a path index. A hash table can be used to store all the elements reachable via a particular path, where the key can be the path itself.

We have found that it is inefficient to evaluate queries with a Generalized Path Expression (GPE) using such an index.

4.2 Value Index

A value index, for a particular attribute, is a storage structure that maintains a set of object references for every value of the attribute. This is useful, as instead of sequentially searching all elements having a particular value or a range of values, a list of elements satisfying the criteria can be directly obtained from the value index.

Query Evaluation in 'Db4XML' is also supported by a value index, which is implemented as a T-tree (Lehman and Carey, 1986) or a B^+ tree (can be chosen as an installation option). Different scopes can be chosen for a value index. A value index can be at a global scope. The advantage of this scheme is its simplicity, but such an index becomes too large. Value index can be constructed for each tag name. For example, the tag 'title' is used by a number of XML

elements. A value index can be constructed for all the attribute values of element nodes having 'title' as an incoming edge. Such an index will list references to all the element nodes with incoming edge 'title' and having a certain value. Such an index helps in evaluating GPEs such as *.title = 'XYZ'. The value index can return all the elements with title 'XYZ'. Finally, for each path in the DOM tree, a value index can be kept. Such an index can be merged with the path index by maintaining a reference to it in the path index.

5 Engine Architecture and Implementation

In this section, we present an overview of the 'Db4XML' engine architecture and summarize the functions of its main components. The 'Db4XML' engine has three main functions - storage of data and metadata, facilitating concurrency and providing an interface to the query processing component.

In our implementation, the data is stored as a collection of `Element` objects. This collection is in the form of a hash table, which is stored in `ElementBag` (Figure 9). `ElementBag` apart from being a container for `Element` objects also holds references to all the other objects in the engine. Meta-data is stored as a collection of `ElementType` objects. This collection is stored as a hash table in `ElementTypeBag.` Apart from holding a reference to `ElementTypeBag`, the `ElementBag` also holds the references to `PathIndex` and `ValueIndexTable.` The `PathIndex` object stores the path index (Patricia Trie) and the `ValueIndexTable` stores references to all the value indices.

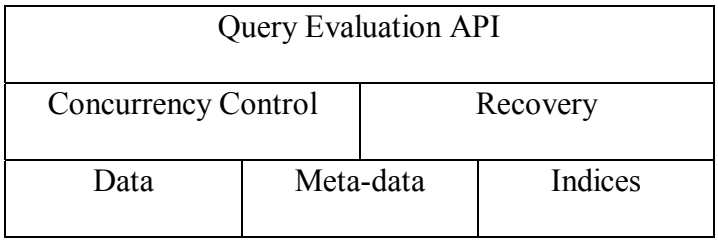


Figure 8. Architecture of the 'DB4XML' engine

In order to facilitate concurrency, there is a `TransactionManager` class that maintains the identifiers of all the active transactions and a list of all the operations performed by them as an Active Transaction Table (ATT). The information about each operation, *i.e.*, type of operation (update, select, insert and delete), after image of the element being accessed is stored as an `Operation` object. In addition, there are two other classes that maintain locking information, `LockTable` and `LockInfo`. `LockTable` maintains a set of all locks currently held on all `Elements` as a collection of `LockInfo` objects. Each `LockInfo` object for an `Element` contains information about the type of the lock being held (read / write), number of reader and writer threads and the list of the transactions that have acquired locks on that element.

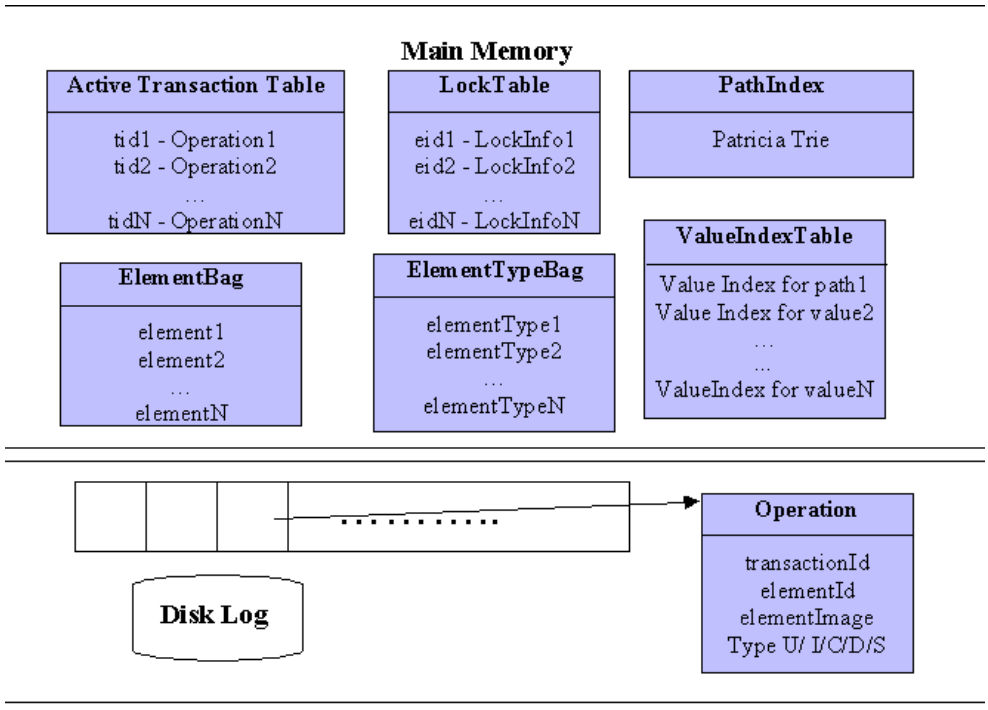


Figure 9. Main memory representation of the 'Db4XML' engine

The 'Db4XML' database engine also provides a remote API for the Query Processing Component*. It is a low–level API, which can execute the logical query plan generated by the Query Processing Component. Our current implementation uses Java RMI for remote method invocation.

6 Query Processing

Query Processing for XML databases is more challenging than for relational databases due to the semistructured nature of XML data. One typically needs to find elements associatively and / or navigate from element to element in very flexible ways.

A number of query languages for XML data has been proposed. A good survey of these lanaguages is done in (Chinwala et al., 2001). Before discussing query processing, the query language XQuery (XQuery 1.0, 2001) is highlighted. XQuery is the W3C recommended language for querying XML databases. It has been derived from XML query language called Quilt (Chamberlin et al., 2000). XQuery has specifically been designed for providing efficient queries for XML documents. It contains XPath (XML Path Language (XPath) Version 1.0, 1999) as a subset. XQuery is a functional language in which a query is represented as an expression. XQuery supports several kinds of expressions, and the structure and appearance of a query may differ significantly depending on which kinds of expressions are used. The principal forms of XQuery expressions are as follows: Path expressions, Element constructors, FLWR (FOR, LET, WHERE and RETURN) expressions, expressions involving operators and functions, Conditional expressions, Quantified expressions and expressions that test or modify data types. In FLWR

expressions, the FOR clause is used to identify the data sources and initialize variables to the entry paths in the data. The LET clause is used to assign variables to different paths in the data. The conditions are specified in the WHERE clause and the RETURN clause specifies the format of the results. There are many features in XQuery, but for conciseness, we will briefly describe a basic query. This query retrieves names and identification number of all the employees, whose salary is less than 50000.

```
FOR $e IN /employees/employee
WHERE $e/salary < 50000
RETURN
      <employee eid = {$e/@id}>
            {<name> { $e/name } </name>}
      </employee>
```

The answer to this query is as follows.

```
<employee eid='777-97-0936'>
      <name>Greg</name>
</employee>
```

'Db4XML' supports FOR, WHERE and RETURN clauses of XQuery (LET is currently under development). XQuery also provides the ability to pose queries with incomplete structural information. This is facilitated by the use of wild cards such as *, ? , *etc*. in the path expressions contained in a query. Wild cards may be used anywhere in the path expression. The symbol '//' stands for sequence of zero or more labels and star ('*') stands for a missing label. A label is same as a tag name in a path expression. In XQuery, wild card may occur anywhere in the FOR clause or the WHERE clause of the query. GPEs are also useful because they offer shortcuts. 'Db4XML' also supports joins using XQuery. Typically, joins are specified by having at least two variables defined in FOR or LET clauses, and then specifying conditions in the WHERE clause, having these variables on both sides.

---

[*] The query processor an the database engine can also be combined as one process.

Naturally, in 'Db4XML', much of query processing is done in the Query Processing Component, which is responsible for parsing the query and generating a syntax tree. The syntax tree is then converted to an algebra tree and the algebra tree is traversed to generate a logical query plan for the engine. The logical query plan is converted to a physical query plan. One novel aspect of our query processor is the use of XSLT (XSLT,1999) to transform the XQuery syntax tree into algebra tree. The focus of this paper is on low level query processing that is part of the database engine, often referred to as query evaluation.

6.1 Query Evaluation

Query evaluation involves executing the user query based on the physical query plan provided by the Query Processing Component. The physical query plan takes advantage of the indices and storage mechanism of the engine. In 'Db4XML', the database engine provides a rich API for evaluating the physical query plan. In our implementation, a number of query evaluation techniques have been implemented and tested. Before discussing these, we briefly review the query evaluation techniques used by Lore, QuiXote, XSet and Niagara.

Lore uses a *recursive iterator* approach for query evaluation (McHugh and Widom, 1999). With *iterators*, query evaluation begins at the top of the query plan, with each node in the plan requesting tuples from its children and performing operations on them. The QuiXote (Mani and Sundaresan, 1996) query processing system consists of two parts, the preprocessor and the query processor. The preprocessor extracts schema information from XML documents and computes structural relationship sets. The sets generated together with indices are used to evaluate queries. XSet (Zhao and Joseph, 1999) uses a structure index (path index) to evaluate

queries. Niagara uses a search engine to retrieve XML files on the Web containing a given element, text or text pattern. It then uses its indices, element and word lists (see section 4), to find relevant elements in the XML documents retrieved.

Significant work has been done on evaluating queries with Generalized Path Expressions (GPEs). Queries with GPEs can be evaluated at run time by following the database graph. This approach can be costly. The cost of evaluating Generalized Path Expressions can be reduced by using partial knowledge of graph's structure (Fernandez and Suciu, 1998). Information about the underlying schema can be used for optimization of such queries. If a database includes structural summary, then the GPE can be expanded using the structural summary (Christophides et al., 1996). In (Fernandez and Suciu, 1998) two optimization techniques, query pruning and query rewriting using state extents are used. An extent is like a class extent of an object-oriented database that references all objects of one type. In query pruning, the search can be pruned at certain subgraphs in the database graph. For example, to evaluate GPE *.department.*, we will explore only under the subgraphs containing department links. In the other technique, the state extents are precomputed and the schema and the query are inspected to find which extents belong to the result. For example, to evaluate GPE *.department.*.project, all the project extents are examined. 'Db4XML' also provides the ability to pose queries with GPE. In 'Db4XML', different techniques have been used to evaluate GPEs.

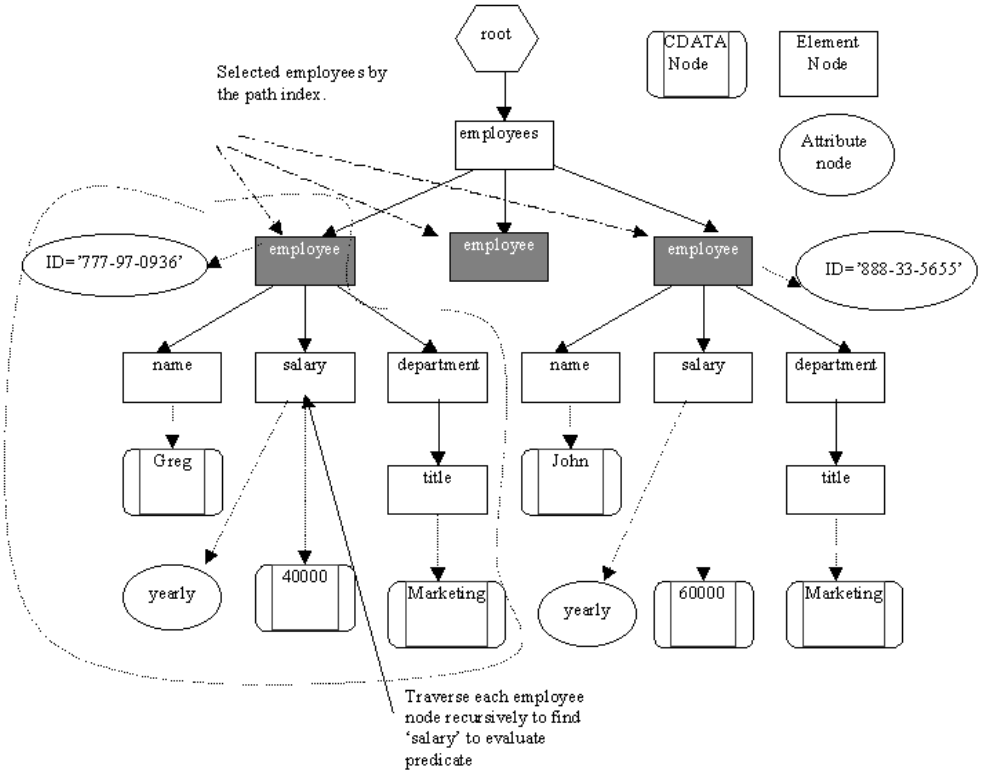
In the rest of this section, we present a comparative study of various query evaluation techniques. First we will summarize various techniques that can be used for evaluating queries on XML data. Consider the query to retrieve information about all employees having salary less than 40000.

```
FOR $e IN /employees/employee
WHERE $e/salary < 40000
RETURN
      <employee eid = {$e/@id}>
```

```
        {<name> { $e/name } </name>}
        {<salary> { $e/salary } </salary>}
</employee>
```

This query can be evaluated by using the following techniques:

- Top down approach - In the top down approach, the database graph is traversed from the root to the bottom and element nodes matching the path in the query are selected. The conditions in the query are applied to the selected nodes.
- Top down approach using path index - In the previous approach, the whole database graph has to be traversed. The path, in the FOR clause of XQuery, can be used to prune the search. For the example query, the path index is used to get a list of all the elements reachable via the path "/employees/employee". The conditions are evaluated by recursively going down the DOM tree for the XML document, starting from each "/employees/employee" node, since it is the path specified in the FOR clause of the query, and finding "/employees/employee/salary" (the path specified in the WHERE clause) nodes, as shown in Figure 10. This is required when no structural information is available beyond the path in the FOR clause of the query ("/employees/employee").
- Bottom up approach using path index – In this approach, the path index is used to find a



list of element nodes reachable via the path in the WHERE clause of the query instead of the path in the FOR clause of XQuery. The list of element nodes can be pruned by applying the conditions on it. Parent pointers can be used to traverse up the database graph, if required. This can eliminate the need for an exhaustive search as done in the top

Figure 10. Top down approach using path index to evaluate query

down approach. In the example mentioned above, "/employees/employee/salary" elements can be located first using the path index. Then parent pointers can be followed to find "/employees/employee" elements.

- Bottom up approach using value index - This method is different from the previous methods because there is no need to prune the element node list, which is obtained by the path index. A value index is used to find the list of element nodes satisfying the condition in the WHERE clause. Again, the parent pointers can be used to match edges backward, if required.

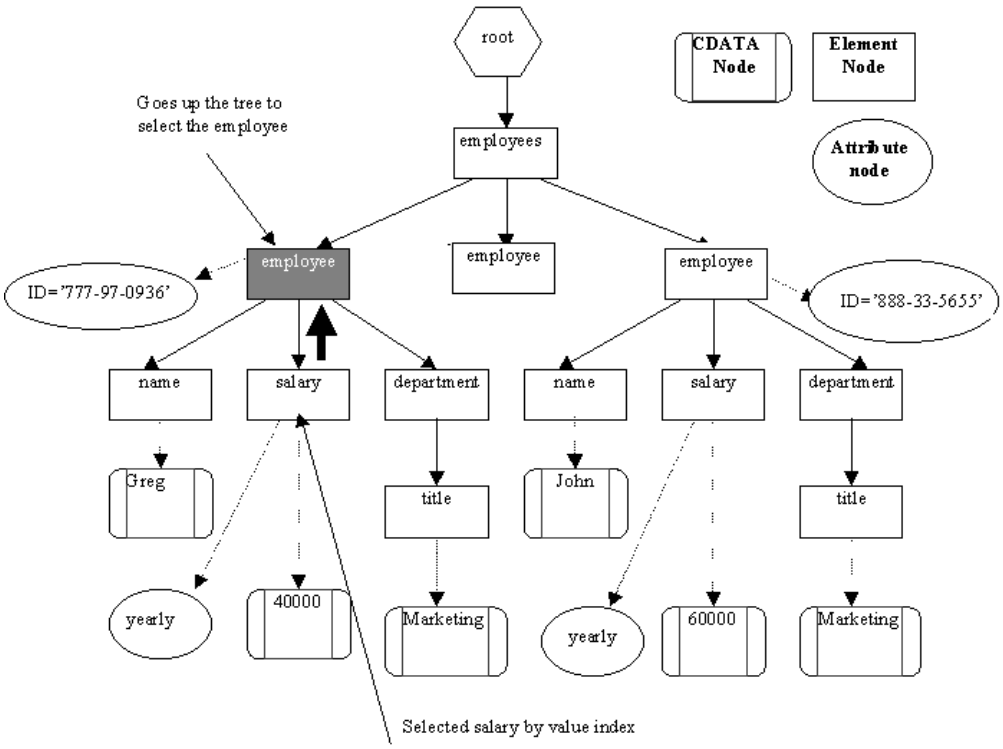


Figure 11. Using value index to evaluate query

The presence of structural information about the XML documents can lead to a number of optimizations in query evaluation. While traversing the database graph, information can be used to find the order and type of children of an element node. For example, while traversing the database graph to find <name> element, which is children of <employee> element, type information can be used to find out that the <name> element is always the first child of the <employee> element. While using the value index based on tag names, type information can be used to find the exact path of the element. This will avoid traversing the database graph up. For example, while evaluating expression [/employees/employee/name = 'Greg'] using a value index on tag 'name', the type information is used to find the exact paths of the elements returned by the value index.

6.2 Performance Study

This section evaluates the performance of various strategies discussed in the previous section on the 'employees' dataset. The 'employees' dataset models an employee database in a company. We have performed tests on incremental sizes of data. 'Db4XML' is a main memory resident database. Suitable index structures are created to support different techniques of query evaluation. Data size is measured by the number of employee records in the 'employees' document.

Our experiments were conducted on a Sun Solaris machine SPARC 5 with 128 MB of real memory. Large datasets required explicit allocation of the heap size using Java non-standard options (Xms, Xmx). The query time measured is the 'elapsed time' (not the 'CPU time'), but the system was dedicated to this study at that time.

The set of experiments has been designed to comprehensively compare various query evaluation techniques. Time measurement has been done for different types of queries such as point, range and set containment queries. Queries with Generalized Path Expressions (GPEs) are also evaluated. Suitable indices are created for different query evaluation techniques. The results of the study have given an insight into the benefits and the shortcomings of the various query evaluation techniques.

The time to execute a query includes the time taken to parse the query by the Query Processing Component, the time taken by the engine to find the list of relevant elements and the time taken to extract required information from the list of the elements. Since we are comparing various query evaluation techniques, we have measured the time taken by the engine to retrieve the list of relevant elements. This is because the time taken to parse the query and the time taken to extract the required information from the list of elements retrieved is same for all the techniques .

6.2.1 Point Query

A point query is the simplest possible query. Three approaches are used to evaluate such a query. A Patricia Trie is used as the path index, while a T-tree is used as the value index. This query finds employee identification number (id) for employee "James Smith'.

Query	FOR $e in employees/employee/name WHERE $e = 'Greg' RETURN $e/@id		
Data Size	**Query Evaluation Techniques**		
	Top down (a)	**Path Index (b, c)**	**Value Index (d)**
10000	245	52	1
20000	308	84	1
30000	689	113	1
40000	710	144	1
50000	850	175	2
60000	1000	210	2
70000	1177	238	2

80000	1300	316	2

Table 3. Query time taken in ms for point query (Query 1)

As seen in the Table 3, the value index approach is the best approach for evaluating point queries. For smaller data sizes, it can be seen that it is efficient to evaluate queries using the path index. Since the paths in the FOR and the WHERE clause are the same, there is no difference in the execution times for the bottom up approach using path index and the top down approach using path index. The top down approach, as expected, has the worst performance.

6.2.2 Range Query

Consider a query to retrieve all the salaries less than 50,000 being paid by the company. To evaluate this query using a value index, a B^+ tree on salary is utilized. B^+ tree is used to evaluate range query because sibling pointers in a B^+ tree can be followed to get results faster (Knuth, 1973).

Query	FOR $e in employees/employee/salary WHERE $e < 50000 RETURN $e		
Data Size	**Query Evaluation Techniques**		
	Top down (a)	**Path Index (b, c)**	**Value Index (d)**
10000	150	37	1
20000	342	70	4
30000	455	104	4
40000	622	139	4
50000	920	174	4
60000	963	217	5
70000	1351	240	5
80000	1569	277	5

Table 4. Query time taken in ms for a range query (Query 2)

Again, since the paths in the FOR and the WHERE clause are the same, there is no difference in the execution times for the bottom up approach using path index and the top down approach using path index (b, c). There is little difference between the query evaluation time of a range query and a point query with the path index approach. This is because the condition is evaluated for same number of elements as in both the cases the same element list is obtained from the path index. The time taken to execute a range query is more than that of a point query using a value index. This is because the value index need to be traversed more to search all the keys within a range of values.

6.2.3 Set Containment Query

Consider a query to list employee identification numbers of all the employees working for the 'Marketing' department. This query requires traversing the database graph (DOM) up or down depending on the query evaluation technique. A value index (T-tree) created separately on title is utilized.

Query	FOR $e in employees/employee WHERE $e/department/title = 'Marketing' RETURN $e/@id			
Data Size	**Query Evaluation Techniques**			
	Top down (a)	**Path Index (b)**	**Path Index (c)**	**Value Index (d)**
10000	183	121	26	1
20000	475	258	63	1
30000	588	382	92	1
40000	930	430	109	1
50000	1182	599	131	2
60000	1201	757	171	2
70000	1310	1236	188	2
80000	1509	1303	225	2

Table 5. Query time taken in ms for a set containment query (Query 3)

As shown in Table 5, the bottom up approach performs better than the top down approach, for set containment queries, while using the path index. Again, the value index approach is the most efficient approach.

6.2.4 GPE Queries in 'Db4XML'

The path index in 'Db4XML' contains structural information of the underlying database. All matching paths to a GPE can be extracted from the path index by matching edges in the GPE while traversing the path index.

We have evaluated queries with GPE using three different techniques. In the top down approach, edges in the path are matched to the tag names of element nodes until a '//' is encountered in the GPE. This is the point where we can begin exhaustive search of the database. In the second approach, the path index (Patricia Trie) is used to expand the GPE and retrieve all the elements having matched paths at the same time. Note that this will eliminate redundant path traversals. The second approach is more useful if the size of structural summary is small (path index). Finally, value indices can be used to get a list of all the elements satisfying the conditions.

For queries having a GPE with a '//' at the beginning, exhaustive search needs to be done if query is evaluated in a top down fashion. The execution of such queries can be made faster by building a value index on the tag name (title in this case).

Query	FOR $t in //title WHERE $t = 'Marketing' RETURN $t		
Data Size	**Query Evaluation Techniques**		
	Top down (a)	**Path Index (b)**	**Value index (c)**
10000	394	49	5
20000	839	96	5
30000	1256	128	5
40000	1684	183	5
50000	1909	177	6
60000	2693	225	6

70000	2852	297	6
80000	2880	347	6

Table 6. Time taken in ms for query containing GPE with a '//' at the beginning

Queries with a '//' in the middle of a GPE can be evaluated in a top down fashion. The Path index can be used to match exact paths and alternation can be used to obtain results. For the query mentioned below, the matched path for the GPE is employees.employee.department.title. Since there is only one path matched, no alternation or union is required. A value index based on the 'title' tag name can speed up the execution considerably.

Query	FOR $t in employees//title WHERE $t = 'Marketing' RETURN $t		
Data Size	**Query Evaluation Techniques**		
	Top down (a)	**Path Index (b)**	**Value index (c)**
10000	229	30	5
20000	347	74	5
30000	728	109	5
40000	896	127	6
50000	1117	158	6
60000	1375	197	6
70000	1659	229	6
80000	1874	259	6

Table 7. Time taken in ms for query containing a GPE with a '//' in the middle

Compile time expansion might result in a smaller number of possible matched paths (which explains faster execution while using the path index to find exact paths).

When evaluating queries with a '//' at end of a GPE, in a top down fashion, recursive search need to be done as soon as a '//' is encountered. When using the path index, all the paths matching the expression can be found. The number of paths could range from a few to a very large number. In the example mentioned in Table 8, the matched paths are employees.employee, employees.employee.name, employees.employee.salary, employees.employee.department,

employees.employee.department.title, *etc*. This can result in slower query evaluation using the path index. We need a value index at global level to evaluate such a query. Such an index may be too large.

Query	FOR $e in employees/employee// WHERE $e = 'Marketing' RETURN $e	
Data Size	**Query Evaluation Techniques**	
	Top down (a)	**Path Index (b)**
10000	169	417
20000	347	846
30000	516	1280
40000	676	1671
50000	831	2021
60000	1020	2613
70000	1232	3377
80000	1426	4000

Table 8. Time taken in ms for query containing a GPE with a '//' at the end

6.2.5 Summary

The graphs in Figure 12 summarize the performance of the various query evaluation techniques. For a point and a range query, the performance is worst, when using the top down approach. This is because one has to traverse the whole database graph to answer a query. The value index approach is the best approach. The path index approach (top down / bottom up) is the next best approach. There is no difference between the top down and the bottom up approach when using the path index because we are retrieving same elements on which the conditions are applied. So the element list given by the path index is pruned by applying the conditions on it and no traversal is done in the database graph.

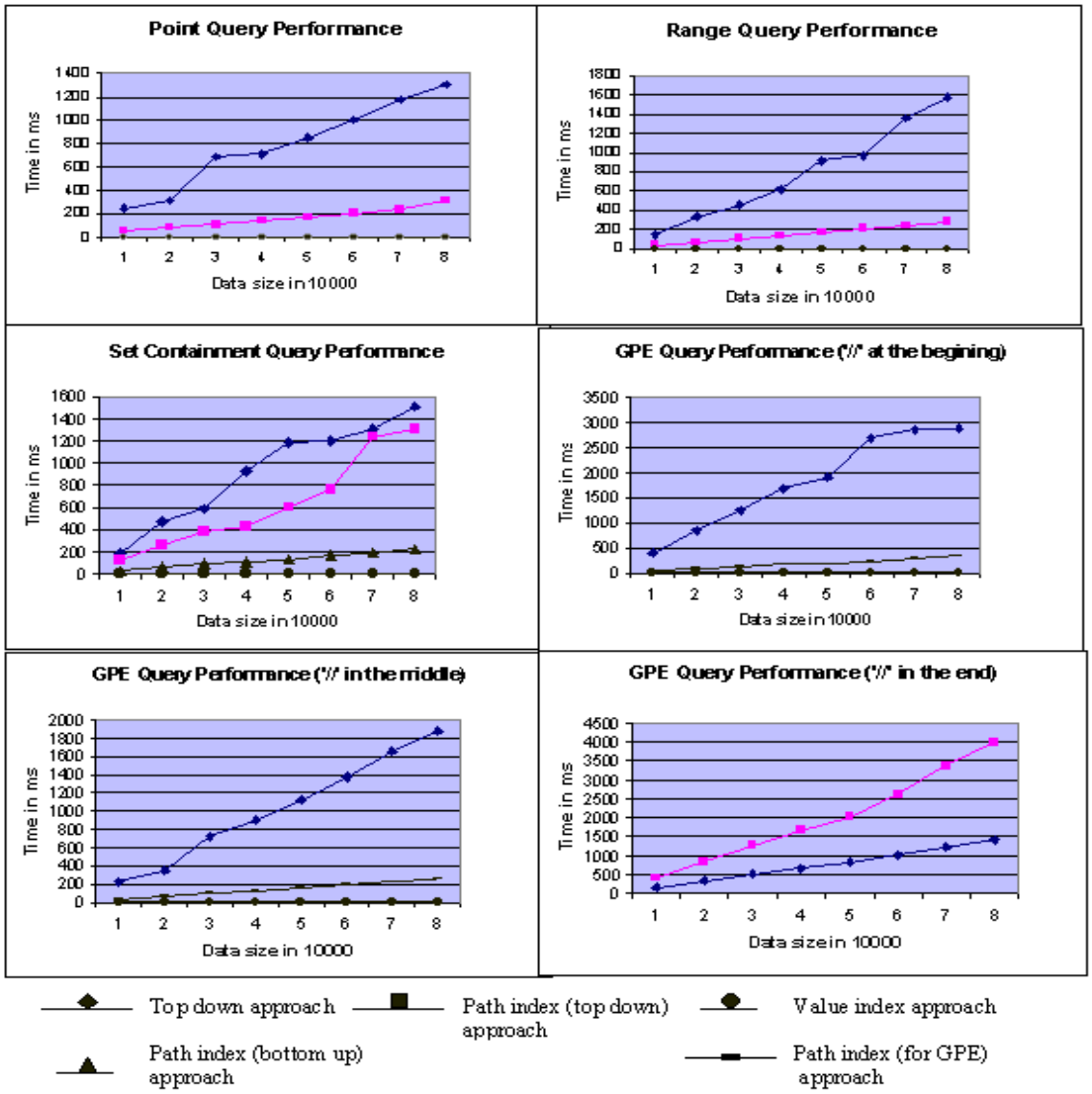


Figure 12. Summary of the performance study of the various query evaluation techniques

For a set containment query, again, the value index approach is the fastest approach. The bottom up approach using path index is remarkably better than the top down approach using path index. This is because no recursive search is required in the bottom up approach using path

index. The conditions can be applied on the list of the elements given by the path index. In the top down approach using path index, it is necessary to recursively descend down the element nodes. So the performance of the path index (top down) approach is only little better than performance of the top down approach.

Figure 12 also shows the summary of performance of a query with a GPE ('//' in the beginning). It can be seen that performance is not acceptable with the top down approach. The performance of the engine is much better when using the path index to expand the GPE. Expanding the GPE, in this case, has resulted in smaller number of matched paths. The value index approach again outperforms other approaches. The performance of the query containing GPE ('//' in the middle) is very similar to the performance of the query containing GPE ('//' in the beginning). This is because of same reasons as mentioned above.

Figure 12 also shows the performance of a query with a GPE ('//' at the end). The performance is worst when using the path index to expand the GPE. This is because, expanding the GPE has resulted in large number of matched paths. It might be better to evaluate such a query using the top down approach.

7 Transaction Management in 'Db4XML'

We have discussed the data model, storage and index structures, query evaluation and architecture of the 'Db4XML' engine. Since our system is a multiple user system, we need to incorporate a concurrency control mechanism. We provide support for transactions (Gray, 1981). Transaction management is required to ensure Atomicity, Consistency, Isolation and Durability (ACID) properties of transactions (Harder and Reuter, 1983). Concurrency control allows multiple users to access the same data. Recovery means the restoration of the database to the most recent consistent state just prior to the time of failure. In this section, we describe the issues

involved in using concurrency control for XML data. It discusses the concurrency control technique and recovery mechanism used by the 'Db4XML' engine. Finally, a performance study of the effect of introducing concurrency control on the various query evaluation techniques is carried out. There are no W3C standards for a Data Manipulation Language for XML data. XUpdate is an effort in that direction (Xupdate, 2000).

7.1 Concurrency Control

In relational databases, all tuples being read or written by a transaction are locked along with the 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 query to retrieve the names of all the employees with salary less than 50,000.

```
FOR $e IN /employees/employee
WHERE $e/salary < 50000
RETURN
      <employee >
            {<name> { $e/name } </name>}
      </employee>
```

To execute this query, a list of all the elements having path '/employees/employee/salary' is retrieved. The list is pruned by applying the conditions on the elements. Then the database graph is traversed to retrieve the 'employees/employee/name' elements. Suppose a transaction T_1 runs this query on the engine. T_1 must hold read locks on all the '/employees/employee/salary' elements which satisfy the condition, their ancestor 'employee/employee' elements and all the '/employees/employee/name' which are part of the result, so that no other transaction T_2 can modify them. This is done to ensure the ACID properties of the transactions.

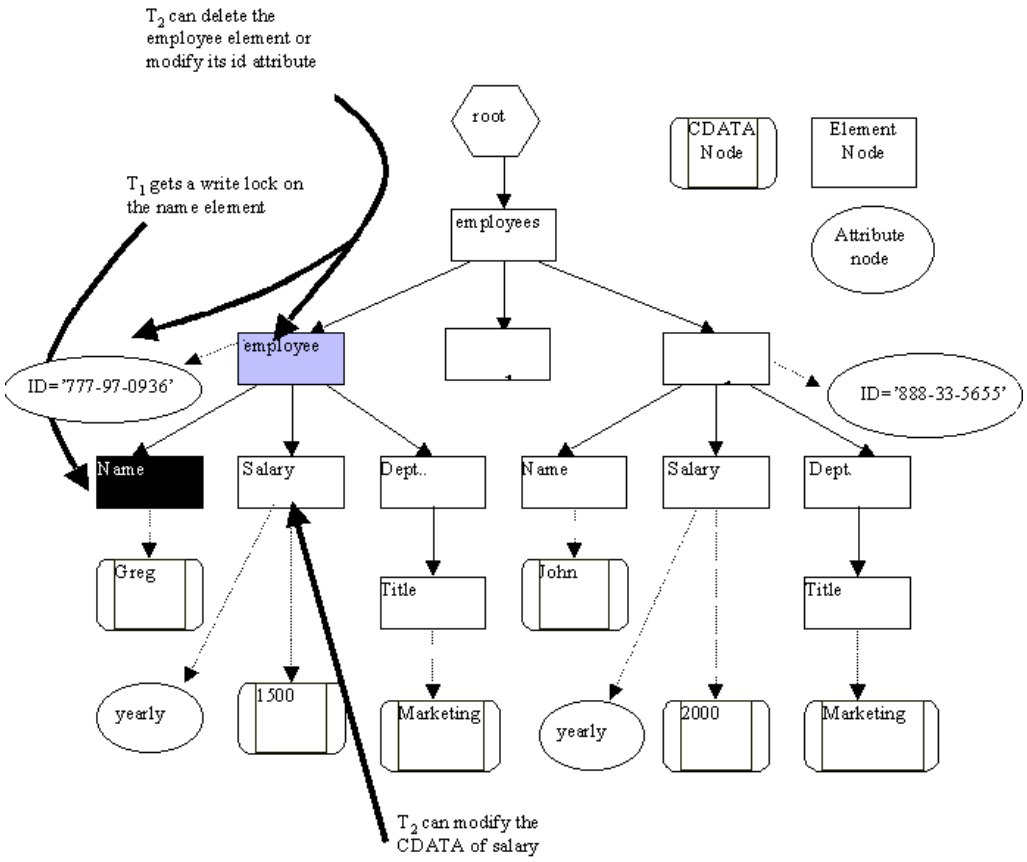


Figure 13. Problems that can occur while locking XML data

In order to avoid concurrency problems, the following rules are used while locking XML data.

- Whenever a transaction reads elements in a top down / bottom up fashion it acquires read locks on element nodes being traversed.
- All elements read or written are locked and the locks are released for those elements, which do not satisfy the conditions. Locks on all the elements which

satisfy the conditions and which are part of the result are retained. As in the example query, locks on element nodes having paths 'employees/employee/salary', 'employees/employee/' and 'employees/employee/name' will be maintained, for all the elements in the employee substructure which satisfy the condition.

- If a transaction deletes an element node, it has to acquire write locks on all the descendants of the 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 further operations in the transaction invoke methods on the engine using the unique identifier (`transactionId`). A transaction has to obtain read or write locks on indices, element nodes and their ancestors, which are being read or written by it. If all the operations are successful, the transaction commits. We have used rigourous-2PL as the locking scheme (Eswaran et al., 1976), as a result of which a transaction releases all its locks only after commit point. The engine uses a deferred update scheme, as a result of which elements are not updated before commit point[*]. Instead an image of the updated element, along with the type of operation is stored in the Active Transaction Table (ATT) for every operation. The engine also maintains a graph of waiting transactions 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

[*] A transaction T reaches its commit point when all its operations have been executed successfully and their record added to the log. After the commit point, the transaction is said to be committed and the effects of transaction are recorded in the database. The transaction then writes [commit,T] entry into the log

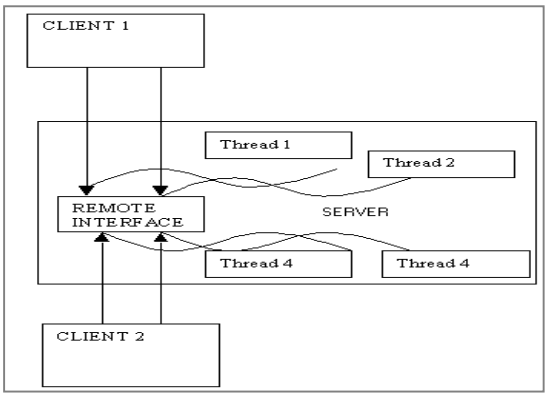


Figure 14. Client Server Architecture for the 'Db4XML' engine

are released and all the instances of the transaction in 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 in the ATT is removed.

7.3 Performance with Concurrency Control

For the previous studies presented in this paper, we have measured the performance of the 'Db4XML' engine without using concurrency control. Concurrency control is an indispensable part of any database system. With a Native XML database, there are issues involved in concurrency control, because of the DOM structure of the XML data (as we have seen

in previous section). It is interesting to look into the effect of concurrency control on the various query evaluation techniques.

In the top down approach for query evaluation, the database graph is traversed from the root to the bottom until useful information is obtained. Hence read locks on each and every element node have to be obtained, while the database graph is traversed. For evaluating queries using the path index, read locks have to be obtained on the nodes of the path index itself. All the elements in the element list returned by the path index need to be locked as they are read and evaluated. If it is necessary to traverse the database graph up or down, read locks are obtained on the element nodes as they are read. While evaluating the query using the value index, read locks are obtained on the nodes of the value index. Read locks are also obtained on all the elements returned by the value index. If the number of elements of a certain type (such as employee elements) is very large, this may induce substantial overhead. This can be observed in the preliminary performance results in which our current concurrency control scheme is used (see Appendix B). We are currently implementing a more efficient concurrency control scheme that is discussed in the future work section.

7.4 Recovery

Recovery means the restoration of the database to the most recent consistent state just prior to the time of failure. In a disk-resident system, users operate on a main memory buffer. When the transactions commit, changes in the database are written to a log on the disk. While, the recovery techniques in both the disk-resident and the main-memory resident systems are similar, their goals are different. A disk-resident system tries to minimize the disk I/O while its counterpart tries to minimize the CPU time (Garcia-Molina and Salem, 1992). The recovery technique used in a database system also depends on the update strategy used (Garcia-Molina and

Salem, 1990). A good model for a recovery technique in the main-memory database systems is presented in Dali (Bohannon et al., 1997). Dali uses a Ping-Pong Checkpointing scheme for recovery in main-memory databases for an immediate update scheme. The 'Db4XML' engine uses a deferred update strategy. Use of deferred update strategy simplifies its recovery scheme and procedure.

The 'Db4XML' engine uses a log on disk, which contains a sequence of operations for all the committed transactions. If operations are being added to the log, the log could be infinitely growing. Hence the engine maintains a database image. The database image is periodically updated. This is done by a separate thread called `Checkpointer`.

During the recovery process the database can be restored from the log. The log can be read serially and all the operations redone. Since we are following the deferred update strategy, no UNDO operations are necessary. To account for the failure, while taking the checkpoint, two database images can be maintained which are updated, alternately. In case of failure while taking a checkpoint, the database can be brought up using the other checkpoint.

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 storage model for the storage of XML documents. The storage model used in 'Db4XML' allows simpler and fewer index structures (e.g. our storage and indexing structures are simpler than those of QuiXote or Lore)

The 'Db4XML' engine provides an API to evaluate a query using multiple techniques. We are currently working on a comprehensive performance study. This paper compares the performance of the various query evaluation techniques. It has been seen that the value index approach for query evaluation outperforms other techniques for most of the cases. In absence of

the value index, the bottom up approach using path index is the next best approach. It can only be used if we have complete structural information (path index) about the path in the WHERE clause of the query. Alternatively, the top down approach using path index can be used. In absence of any structural information, the database graph has to be traversed from the root to the bottom until useful information is found (top down approach). As we have seen, evaluating a query in such fashion consumes more time.

A study has also been done on queries with Generalized Path Expressions (GPEs). We have seen that the query evaluation technique depends on the position of the wild card character in the GPE. If the wild card character is present at the beginning or in the middle of the GPE, constructing a value index for the tag name, which constitutes the last part of the GPE, can reduce the query execution time. The next best approach is using the path index to expand the GPE. If the wild card is present at the end of the query we need a value index at the global level. Such an index may be too large in general. We have seen that using a path index to expand the GPE can result in a number of subqueries, which may be inefficient to evaluate. It might be worthwhile to evaluate the query using the top down approach.

The 'Db4XML' engine provides basic concurrency control and recovery mechanism. We have addressed the problems that could occur while using locks on XML data with DOM structure. We have implemented concurrency techniques, which can be used for the various query evaluation techniques. Preliminary results show a significant degradation in performance when concurrency control is introduced in the various query evaluation techniques. We are currently implementing more efficient techniques for concurrency control. In our present technique, we acquire locks on all the elements of one type, which are given by the path index. This may not be required, if while using alternative query evaluation technique (value index, *etc*.), we acquire locks on the nodes of the path index (which corresponds to all elements of one type). While

traversing the database graph, we can get the type information about the children / parent of an element node. Before reading the children / parent, type information can be used to get locks on the nodes of the path index (which corresponds to elements of one type).

Some new functionality is to be added to the 'Db4XML' engine. We plan to incorporate text searching capability in the database including the use of text indices. We will be adding support for LET clauses in FLWR expressions of XQuery. We also intend to provide security mechanisms in 'Db4XML' in a future version. We are also progressing towards the full implementation of our proposed recovery scheme.

Acknowledgements

We are thankful to Drs. Suchi Bhandarkar and I. Budak Arpinar for reviewing this work and offering their valuable suggestions.

References

Arpinar, I.B., Miller, J.A. and Sheth, A.P., 2001. An efficient data extraction and storage utility for XML documents, Proceedings of the 39th Annual ACM Southeast Conference. Athens, Georgia, pp. 293-295.

Bohannon, P., Rastogi, R., Lieuwen, D., Seshadri, S., Silberschatz, A. and Sudarshan, S., 1997. The architecture of the Dali main-memory storage manager, Journal of Multimedia Tools and Applications 4(7), pp.115-151.

Chamberlin, D., Robie, J. and Florescu, D., 2000. Quilt: an XML query language for heterogeneous data sources, Proceedings of the Third International Workshop on the Web and Databases (WebDB 2000), Dallas, Texas, pp. 53-62.

Chinwala, M., Malhotra, R. and Miller, J.A., 2001. Progress towards standards for XML databases, Proceedings of the 39th Annual ACM Southeast Conference, Athens, Georgia, pp. 277-284.

Christophides, V., Cluet, S. and Moerkotte, G., 1996. Evaluating queries with generalized path expressions, Proceedings of SIGMOD 1996, Montreal, Quebec, Canada, 25(2), pp. 413-422.

Cooper, B.F., Sample, N., Franklin, M.J., Hjaltason, G.R. and Shadmon, M., 2001. A fast index for semistructured data, Proceedings of the 27th International Conference on Very Large Databases (VLDB 2001), Rome, Italy, pp. 341-350.

Document Object Model (DOM), 2002. DOM Level 3 Core Specification, Version 1.0, available at http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020409/.

Eswaran, K.P., Gray, J., Lorie, R.A. and Traiger, I.L., 1976. The notions of consistency and predicate locks in a database system, Communications of the ACM (CACM), 19(11), pp. 624-633.

eXtensible Information Server (XIS)., 2002. available at http://www.exceloncorp.com/products/xis/.

eXtensible Markup Language (XML) 1.0 (Second Edition), 2001. W3C Recommendation, http://www.w3.org/TR/REC-xml.

Fernandez, M.F. and Suciu, D., 1998. Optimizing regular path expressions using graph schemas, Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, pp. 14-23.

Florescu, D. and Kossman, D., 1999. A performance evaluation of alternative mapping schemes for storing XML data in a relational database, Rapport de Recherche, techreport No. 3680 INRIA, Rocquencourt, France.

Garcia-Molina, H. and Salem, K., 1990. System M: A transaction processing testbed for memory resident data, IEEE Transactions on Knowledge and Data Engineering 2(1), pp. 161-172.

Garcia-Mollina, H. and Salem, K., 1992. Main memory database system: An overview, IEEE Transactions on Knowledge and Data Engineering, 4(6), pp. 509-576.

Gray J. 1981. The transaction concept, virtues and limitations (Invited Paper). Proceedings of the Seventh International Conference of Very Large Data Bases (VLDB 1981), September 9-11, Cannes, France, IEEE Press, pp. 144-154.

Gray, J. and Reuter, A., 1993. Transaction Processing: Concepts and Techniques. Morgan Kaufmann.

Harder, T. and Reuter, A., 1983. Principles of transaction oriented database recovery. ACM Computer Surveys 15(4), pp. 287-317.

Knuth, D.E., 1973. The Art of Computer Programming, Volume III: Sorting and Searching, Addison-Wesley.

Lehman, T.J. and Carey, M.J., 1986. A study of index structures for main memory database management systems, Proceedings of the Twelfth International Conference on Very Large Data Bases (VLDB 1986), August 25-28, Kyoto, Japan, pp. 294-303.

Mani, M and Sundaresan, N., 1996. Query processing using QuiXote, IBM research report, available at www.cs.ucla.edu/~mani/xml/ibm.ps.

McHugh, J., Abiteboul, S., Goldman, R., Quass, D. and Widom, J., 1997. Lore: A database management system for semistructured Data. SIGMOD Record, 26(3), pp. 54-66.

McHugh, J., Widom, J., Abiteboul, S., Luo, Q. and Rajaraman, A., 1998. Indexing semistructured data. Technical Report, Stanford University, available at http://www-db.stanford.edu/lore/pubs/index.html.

McHugh, J., and Widom J., 1999. Query optimization for XML, Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases (VLDB 1999), Edinburgh, Scotland, September 7-10, pp. 315-326.

Miller, J.A., and Sheth, S., 2000. Querying XML documents. IEEE Potentials 19(1), pp. 24-26.

Naughton, J., DeWitt, D., Maier D., 2000. The Niagara internet query system (overview paper), available at http://www.cs.wisc.edu/niagara/papers/NIAGRAVLDB00.v4.pdf.

Shanmugasundaram, J., Gang, H., Tufte, K., Zhang, C., DeWitt, D. and Naughton, J.F., 1999, Relational databases for querying XML documents: limitations and opportunities, Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases (VLDB 1999), Edinburgh, Scotland, September 7-10, pp. 312-314.

Sipani, S., Verma, K., Chandrasekaran, S., Zeng, X., Zhu, J., Che, D. and Wang, K., 2002. Designing and XML database engine: API and performance, Proceedings of the 40th Annual ACM Southeast Conference. Raleigh, North Carolina, pp. 239-245.

The Tamino XML Server., 2002. available at http://www.softwareag.com/tamino/.

Tian, F., DeWitt, D., JChen, J. and Zhang, C., 2001. The design and performance evaluation of alternative XML storage strategies, available at http://www.cs.wisc.edu/niagara/Publications.html.

XML Path Language (XPath) Version 1.0., 1999. W3C Recommendation. available at http://www.w3.org/TR/xpath.html .

XSLT.,1999. XSL Transformations Version 1.0, W3C Recommendation, available at http://www.w3.org/TR/xslt.

XQuery 1.0., 2001. An XML Query Language. W3C Working Draft. available at http://www.w3.org/TR/2002/WD-xquery-20020430/.

Xupdate., 2000. XUpdate working draft. available at http://www.xmldb.org/xupdate/xupdate-wd.html.

Zhao B.Y., Joseph, A.D., 1999. XSet: A Lightweight Database for Internet Applications. Master's thesis, Computer Science Division, University of California, Berkeley.

Vitae

Sudhanshu Sipani is a Master's student at the Computer Science Department at the University of Georgia. He has done his M.B.A from Institute of Management Studies, Indore, India and earned his bachelor's degree in Mechanical Engineering. His research interests are XML database systems and real time simulation.

Kunal Verma is a Graduate Student in Computer Science at the University of Georgia. He did his Bachelor's of Engineering in Electronics and Telecommunications Engineering from the University of Bombay, India. His research interests are distributed computing, operating systems, web services, databases and image processing. He has also taught an introductory level course in Java at the University of Georgia.

John A. Miller is a Professor of Computer Science at the University of Georgia and is also the Graduate Coordinator for the department. His research interests include database systems, simulation and workflow as well as parallel and distributed systems. Dr. Miller received the B.S. degree in Applied Mathematics from Northwestern University in 1980 and the M.S. and

Ph.D. in Information and Computer Science from the Georgia Institute of Technology in 1982 and 1986, respectively. During his undergraduate education, he worked as a programmer at the Princeton Plasma Physics Laboratory. Dr. Miller is the author of over 70 technical papers in the areas of database, simulation and workflow. He has been active in the organizational structures of research conferences in all three areas. He is an Associate Editor for *ACM Transactions on Modeling and Computer Simulation* and *IEEE Transactions on Systems, Man and Cybernetics* as well as a Guest Editor for the *International Journal in Computer Simulation* and *IEEE Potentials*

Boanerges Aleman-Meza is a doctoral student in the Computer Science Department of the University of Georgia. He has an Applied Math. Master's degree also from the University of Georgia. His bachelor's degree in computer science is from the Technological Institute of Chihuahua II, Mexico. His research interests are XML, databases and the semantic Web.

Appendix A : XML Schema for 'employees' Document

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

 <xs:element name="employees">
  <xs:complexType>
   <xs:sequence>
     <xs:element ref="employee" minOccurs='1' maxOccurs='unbounded'/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="employee">
  <xs:complexType>
   <xs:sequence>
     <xs:element ref="name"/>
     <xs:element ref="salary"/>
     <xs:element ref="department"/>
   </xs:sequence>
   <xs:attribute name="id" type=”xs:string” use=”required”/>
   <xs:attribute name="supervisor" type=”xs:string” use=”required”/>
  </xs:complexType>

  <keyref name=”employeeKeyRef”>
```

```
  <selector xpath=”/employees/employee”/>
  <field xpath=”@supervisor”/>
  </keyref>

  <key name=”employeeKey”>
  <selector xpath=”/employees/employee”/>
  <field xpath=”@id”/>
  </key>
 </xs:element>

 <xs:element name="name" type="xs:string"/>

 <xs:element name="salary" type="xs:integer"/>
   <xs:complexType>
     <xs:attribute name="payperiod" type=”xs:string”/>
   </xs:complexType>
 </xs:element>

 <xs:element name="department">
  <xs:complexType>
   <xs:sequence>
     <xs:element ref="title" minOccurs='1' maxOccurs='unbounded'/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="title" type="xs:string"/>
</xs:schema>
```

Appendix B : Effect of Concurrency Control on Query Evaluation

Consider the query to find employee identification number of employee ‘Greg’. While using the path index, the path index needs to be locked. Read locks are obtained on all the elements returned by the path index (a T-Tree is used as a value index). While evaluating using the value index, nodes that are traversed in the value index are locked and then read locks are obtained on all the elements returned by the value index.

Query 1	FOR $e in employees/employee/name WHERE $e=’Greg’ RETURN $e/@id	
Data Size	**Query Evaluation Technique**	
	Path Index (b) (c)	**Value Index(d)**
10000	356 (6.84)	2 (2)
20000	520 (6.19)	2 (2)
30000	660 (5.84)	2 (2)

Table 9. Query time taken with concurrency control for point query

It can be seen that the performance degrades more with the use of the path index. This is because we need to lock each and every element obtained from the path index before reading it. While, in the case of the value index, the extra time taken is the time to lock the nodes of the value index itself. This emphasizes the importance of a value index.

For a set containment query, while using the path index, the path index needs to be locked. Read locks are obtained on all the elements returned by the path index. As we have seen, using the top down approach using path indextechnique prunes the search to certain subgraphs in the database graph (all subgraphs under '/employees/employee' element nodes in this example). So read locks may have to be obtained on all the elements in those subgraphs since recursive search is done. In the bottom up approach (with path index), read locks are obtained on the element nodes as the database graph is traversed up. While using the value index, nodes that are traversed in the value index are locked and then read locks are obtained on all the elements returned by the value index. As in all cases, locks are retained on all the element nodes that satisfy the conditions and which are part of the result.

Query	FOR $e in employees/employee WHERE $e/department/title='Marketing' RETURN $e/@id		
Data Size	**Query Evaluation Technique**		
	Path Index (top down) (b)	**Path index (bottom up) (c)**	**Value Index (d)**
10000	1146 (9.47)	225 (8.65)	5 (5)
20000	2200 (8.5)	450 (7.1)	13 (13)
30000	3000 (7.85)	617 (6.7)	27 (27)

Table 10. Query time taken with concurrency control for set containment query

Table 9 shows the time taken by the various query evaluation techniques while using concurrency control. The perormance degradation factor* is in parenthesis. It can be seen that the better the query evaluation technique, the better is its absoulute performance but worse is its relative performance. The absolute performance of the engine is much better for the bottom up approach than for the top down approach using path index. This is because less number of elements have to be locked while using the bottom up approach using path index. The performance of the value index approach is not good because locks are obtained on all the elements given by the value index (1000s in this example)

List Of Tables

List of Figures

* ratio of time taken while using concurrency control and time taken while not using concurrency control