Abstract

XML Database Engines

Rakesh Malhotra and John Miller

Department of Computer Science

University of Georgia

Athens, GA 30602

XML databases lie at the conjunction of two disparate themes and have to address issues related to both. On one hand there is the traditional database field of structured data, query languages, algebras, logical and physical plans while on the other hand is the new standard of the eXtensible Markup Language (XML) for storing and disseminating semi-structured data over the web. This paper focuses on issues involved in implementing the query engine for a native XML database. The issues of data storage, indexing, and query evaluation are discussed in relation to existing XML database systems such as Lore, XSet, and QuiXote. The query engine of MMXDB, a main memory XML database system being developed by us is also described.

INDEX WORDS: Databases, eXtensible Markup Language, Query evaluation, Indexing, Storage, XML, Semi-structured data.

XML DATABASE ENGINES

By

Rakesh Malhotra
rakesh_malhotra@hotmail.com

and

John A. Miller
jam@cs.uga.edu

Department of Computer Science
The University of Georgia
Athens, GA 30602
Phone (706)-542-3440
Fax (706)-542-2966

1. Introduction

Databases, of some form or other, have been with us from the 1960s when the Network (CODASYL) and Hierarchical (IMS) databases first appeared. With the introduction of the relational data model, proposed by Codd (1970), these were replaced by the Relational Database Systems (RDBMs). Since then, RDBMs and the Structured Query Language (SQL) have been the standard database system and query language. The last decade saw the advent of Object Oriented Database Systems (OODBMs) that store objects rather than tuples and Object Relational Database Systems (ORDBMs) that attempt to provide the best of both worlds. However, in recent times, due to the coming of age of the internet, interest has increased in new types of database systems called XML databases. The main difference between the relational/object and XML databases is that while relational/object databases were designed to store structured data, XML databases are designed to store semi-structured data. Semi-structured data can be defined as data whose structure is not rigid, complete, or fixed as required by traditional databases (Abiteboul *et al.*, 1997). XML databases lie at the conjunction of two disparate themes and have to address issues related to both. On one hand there is the traditional database field of structured data, query languages, algebras, logical and physical query plans while, on the other hand is the new standard of the eXtensible Markup Language (XML) for storing and disseminating semi-structured data over the web.

This paper focuses on research conducted on native XML databases with an emphasis on query evaluation strategies, and storage and indexing issues. A native XML database is defined as a system that is developed, from querying to storage, for XML data. The logical and physical data structures for such a system are closely associated with the Document Object Model (DOM), the standard tree based logical data structure for XML data.

The development of emerging database systems for semi-structured data is strongly influenced by the expertise developed for storing and querying structured data. The main components of the query processing system of a database are 1) a parser that generates a syntax tree, 2) an optimizer uses the syntax tree to generate a query plan, taking into account the access structures, as well as the knowledge of the data and the system, and 3) an engine that evaluates the optimized query plan (Abiteboul *et al.*, 1999).

We describe query engines of three XML database systems Lore (McHugh *et al.*, 1997), XSet (Zhao and Joseph, 1999) and QuiXote (Mani and Sundaresan, 2000). In addition, we discuss the implementation of the query engine for a main memory XML database system called MMXDB that we are currently developing. Main memory databases differ from disk databases in that data is completely (or almost completely) resident in shared main-memory. Such databases provide better performance since they eliminate most disk related processing. The details of the parser and optimizer for MMXDB are described in Chinwala and Miller (2001).

The outline of the paper is as follows. After giving a working example, we provide a brief description of MMXDB in section 2. Section 3 deals with several issues about storing and indexing XML data, and evaluating queries on these datasets. Section 4 is a brief discussion of other issues such as transactions, recovery, and security. We conclude the paper in section 5.

1.1 Working Example

The following example of two XML documents staff.xml and departments.xml is used throughout this paper. An employee, in the staff document, has a name, ssn, salary, department number (dno) and office attributes. Office consists of a building and room number. A department has a department name (dname), department number (dno), and the ssn of the manager (mgrssn).

```
<staff>
   <employee ssn = “28656667”>
      <name>Smith</name>
      <salary>28000</salary>
      <dno>28</dno>
      <office><building>A</building><room>6</room></office>
   </employee>
      :
   <employee ssn = “12345678”>
      <name>Clark</name>
      <salary>18000</salary>
      <dno>18</dno>
      <office><building>A</building><room>7</room></office>
   </employee>
</staff>
```

(staff.xml)

```
<departments>
    <department>
        <dname>MIS</dname>
        <dno>28</dno>
        <mgrssn>12345678</mgrssn>
    </department>
       :
    <department>
```

(department.xml)

A Data Type Definition (DTD) for XML defines the document structure and possible elements. For example, based on the (DTD) for staff.xml shown below, it can be inferred that the document can contain zero or more employees and every employee has a name, ssn, salary, dno, and office elements.

```
<!DOCTYPE staff[
        <!ELEMENT staff (employee*)>
        <!ELEMENT employee (name, salary, dno, office)>
        <!ATTLIST employee ssn CDATA  #REQUIRED>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT ssn (#PCDATA)>
        <!ELEMENT salary (#PCDATA)>
        <!ELEMENT dno (#PCDATA)>
        <!ELEMENT office(building, room)>
        <!ELEMENT building (#PCDATA)>
        <!ELEMENT room (#PCDATA)>
        ]>
```

(DTD for staff.xml)

XML schema is a used to represent the structure of the XML document as an XML document. In addition to the information provided in the DTD, the XML schema also specifies whether the elements are string or integer type. The XML schema for staff.xml is shown below:

```
<xs:group name = “Staff”>
   <xs:element name = “staff”>
      <xs:complexType>
            <xs:group ref = “Employee”
               minOccurs=“1” maxOccurs=unbounded/>
      </xs:complexType>
  </xs:element>
</xs:group>

<xs:group name = “Employee”>
  <xs:element name = “employee”>
   <xs:complexType>
            <xs:attribute name “ssn” type = xs:integer/>
            <xs:element name “name” type = xs:string/>
            <xs:element name “salary” type = xs:integer/>
            <xs:element name “dno” type = xs:integer/>
```

```
            <xs:complexType>
               <xs:group ref = “Office”>
            </xs:complexType>
    </xs: complexType>
  </xs: element>
</xs:group>
<xs:group name = “Office”>
   <xs:element name = “office”>
      <xs: complexType>
            <xs:element name “building” type = xs:string/>
            <xs:element name “room” type = xs:integer/>
      </xs:complexType>
    </xs:element>
</xs:group>
```

(schema for staff.xml)

The DOM, depicted graphically, following the conventions of the XML data model (Fernandez *et al*. 2001), for staff.xml is shown in Figure 1. DOM is a tree structure that retains information about the document hierarchy with paths from the root to all the elements. The DOM forms the basis for the logical storage model for many of the systems discussed in this paper. As can be seen from Figure1, staff is a collection of employees and each employee as five elements.

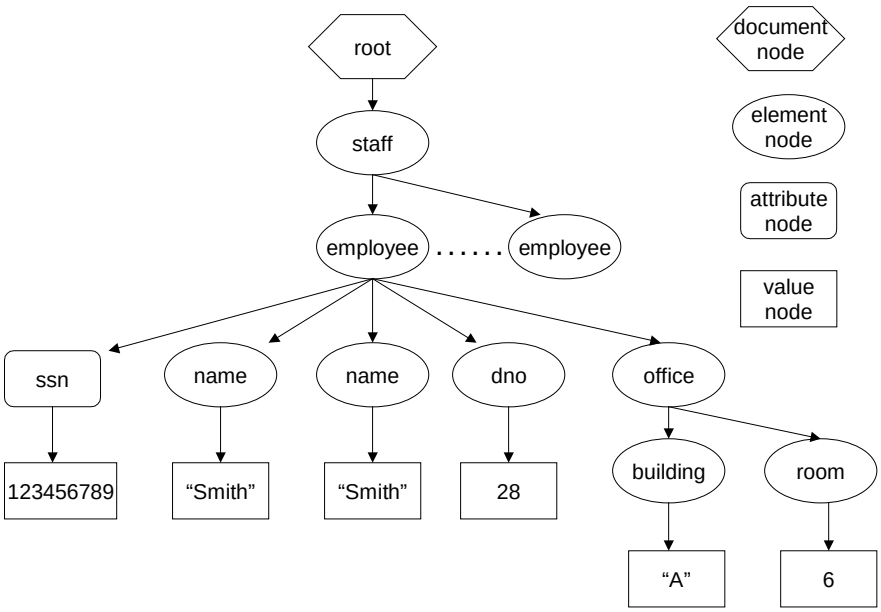

Figure 1. The XML Data Model for staff.xml.

It is also important to choose appropriate data structures to represent XML documents. While staff.xml can be represented using a tree, a collection of staff.xml and department.xml could be stored as a forest of trees. Information contained in the two documents, staff.xml and department.xml can also be combined into a single tree. One of the ways in which the two can be combined is by including each employees department information within the employee tag. If, however, the department and employee data are maintained by separate departments, it may not be desirable to combine them. Also, if a document contains intra-document links, it may be represented using a directed acyclic graph (DAG).

The two example queries used in this paper are:

Q1: Find all employees whose name is Smith.

Q2: Find the name of the manager who manages the MIS Department.

2. MMXDB

The overall architecture of our prototype XML database system MMXDB is shown in Figure 2. It is a main memory database system in the initial stages of development and does not yet have support for transactions, concurrency control, security, or recovery.

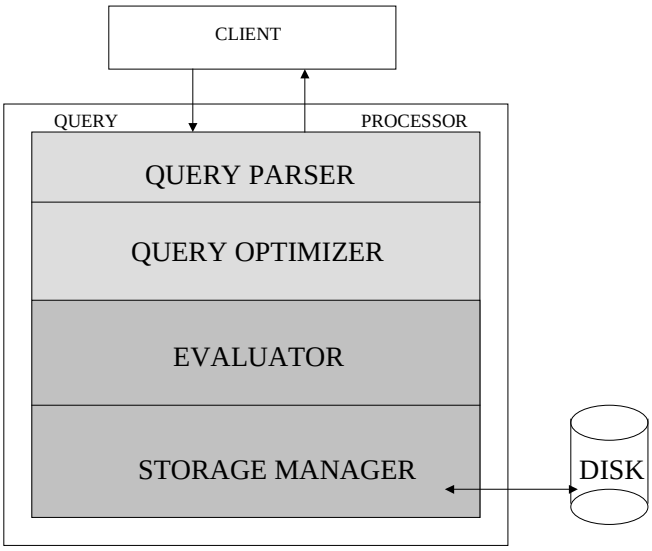

Figure 2. MMXDB – A Native XML database.

We have a thin client which passes user queries in an algebraic form to the database system. The XML algebra of Fernandez *et al*. (2001) whose syntax resembles that of a high level query language was used to specify user queries as well as define the schema and populate data. This algebra was used since it is proposed as a standard for XML algebras by the World Wide Web Consortium (W3C). For a detailed discussion of various XML query languages and algebras refer to Chinwala *et al.* (2001) and Sheth (1999). The query parser and optimizer produce the query evaluation tree. This query evaluation tree is input to the query evaluator which interacts with the storage manager and evaluates the query. The storage manager is responsible for storing, accessing, and indexing the data. This paper focuses on the functionality of the query evaluator and storage manager.

3. XML Database Engines

Three XML database systems – Lore, XSet, and QuiXote were chosen as representative XML systems. Lore (Lightweight Object Repository) which was under development at Stanford University since 1995 morphed into a full featured XML DBMS in the later years (McHugh *et al.*, 1997). XSet is a main memory hierarchically structured database with partial ACID properties and was developed at the University of California at Berkeley. XSet does not support transactions and provides atomicity at the level of individual operations only. This is because it is targeted for the low latency, soft consistency information management applications such as searchable email clients, user preference registries, and online customized content portals. Although it uses a simple query model, it is complete for its current set of target applications (Zhao and Joseph, 1999). QuiXote is the latest XML query processing system being developed by IBM Systems. QuiXote is a two part system where the first part precompiles structural relationship information from the schema and generates indices whereas the second part processes the user query and generates the results (Mani and Sundaresan, 2000). The core issues of developing XML Query Engines (storage, indexing, and query plan evaluation) are discussed with a focus on these systems.

3.1 Storage

3.1.1 Introduction

When large numbers of documents are stored in XML databases and queried on, efficient storage becomes critical. This is because inefficient storage slows down query performance and impedes the system by wasting memory resources. Thus, the three main storage issues are 1) efficient retrieval of data, 2) efficient use of memory resources, and 3) efficient updates to data and schema. Prior to discussing these issues for the XML databases chosen, it is pertinent to address them in relation to the storage techniques used by XML database systems.

Mapping XML data to an existing relational/object-relational/object database: Several studies have focused on mapping XML data to existing relational (Deutsch *et al*., 1998; Shanmugasundaram *et al.*, 1999; Fernandez *et al*., 2000; Miller and Sheth, 2000) object-relational (Klette and Meyer, 1999), and object databases (Tian *et al*., 2000) and several of the present day commercial systems such as Oracle9i (object relational) and IBM's DB2 (object relational) use this technique. The main advantage of using such commercial database systems is that the customer can rely on a familiar vendor, its technology, licensing, and support. Also, XML data can be queried using the popular SQL syntax eliminating the need to learn a new query language. In addition, the other advantages are that most existing database systems provide features such as secondary storage, concurrency control and recovery. However, such systems, particularly the ones based on relational databases also have certain drawbacks like simple queries requiring several expensive joins, and problems with handling complex recursion (Shanmugasundaram *et al.*, 1999). Problems associated with mapping XML data into object-relational databases and some solutions are addressed by Klette and Meyer (1999). Object-oriented databases fare better than their relational / object-relational counterparts. The main advantage of the object approach is that the tree structure of XML documents lends itself well to objects, allowing the system to optimize both storage and query processing (Abiteboul *et al.*, 1999). Each element can be stored as an individual object or as a sub-object inside the document object. The disadvantage is that this approach is not well suited for data whose schema is frequently updated because existing objects have to be modified and new objects have to be created. In addition, if

the creation of objects is taken to the leaf level, *i.e.*, each node in the DOM is an object, it leads to data fragmentation which not only wastes storage space, but also slows down the system.

Storing XML data as a large text files (with or without compression): This is considered the simplest way to store XML data. Prior to query execution, the document is read, parsed, and stored in a DOM like tree in main memory. The advantage of such a system is that it is easy to implement and does not require a database system to store the information. Also there are no reconstruction costs in creating the original document (Tian *et al*., 2000). A major drawback of this approach is that the entire dataset has to be loaded into the main memory before processing a query. This is not a problem for smaller documents but quickly becomes an issue with large documents or a huge collection of smaller documents. Two different techniques, one based on partial retrieval and the other on partial retrieval and compression have been proposed to overcome this drawback. The first method, suggested by (Tian *et al*., 2000), is to create external indices on the XML document. Such indices would store offsets of XML elements inside the text file to assist in the retrieval of partial documents. A similar retrieval approach is proposed by the Millau system (Girardot and Sundaresan, 2000). However, in addition to allowing partial retrieval of documents, Millau also compresses disk resident XML documents by about 80 percent.

Storing the XML in native format: When relational or object oriented database systems map XML data to into their structures, they introduce additional layers between logical data and physical storage slowing down updates and query processing (Kanne and Moerkotte, 2000). Efficient storage of XML data in native form is discussed by Kanne and Moerkotte (2000). Their system, NATIX, supports tree-structured objects at the logical and storage level.

This system's storage manager consists of the "classical physical records manager" and the tree storage manger. The records manger handles disk memory and buffering and provides memory space divided into a collection of equal sized pages. Pages, in turn, can hold records with each record identified by a (pageid, slot) pair. This is also know as a record ID or RID. On top of this records manager, NATIX has a tree storage manager that maps the sub-trees from the XML document into records.

The logical data tree, similar to the one created using DOM, is mapped to the physical data tree. The logical and physical trees are shown in Figure 3. Besides nodes from the logical tree, a physical data tree contains additional nodes that are used to manage large trees. Large trees are defined as trees that cannot be stored on one disk page. Physical nodes can be of three types 1) aggregate nodes which represent the inner nodes (nodes without attributes or values) from the logical tree and can contain their children nodes, 2) literal nodes that contain byte streams representing text, graphic, *etc*., and map the leaf nodes of the logical tree, and 3) proxy nodes that are used by the system to connect one record to another in order to maintain connectivity across pages.

Physical tree nodes are stored in file records (Figure 3). Records contain sets of nodes. The upper limit on the record size is the page size. Large documents that cannot fit on single pages and are split over several records. The document is split into subtrees and each subtree is stored on a single page. Proxy nodes are used to store information about hierarchical connections between subtrees. Helper nodes help the proxy nodes to group all children together. Each proxy node contains the RID of the subsequent record that contains the subtree that fits below that node. Thus, “the decisions about which parts of a document reside on the same page are based on the semantics of the data.” The authors also discuss algorithms for the dynamic maintenance of the storage in case information is added or deleted from the document. Thus, native storage of XML data provides for 1) efficient retrieval of data, 2) efficient use of memory resources, and 3) efficient updates to data. Tamino, developed and marketed by Software A.G., is an example of a commercial native XML database system.

3.1.2 Lore

Lore is a complete database system where the data graph (similar to the DOM) is physically stored on disk. The vertices in the graph are represented as separate objects with unique identifiers. Objects are stored physically on disk pages with multiple objects per page and large objects span multiple pages. Also objects are clustered on a page in a depth first manner based on their location in the data graph. The indexing and query evaluation techniques which we discuss later are for this implementation of Lore. Later, in conjunction with the development of Lore, Ozone, a system for storing data in Object

Exchange Model (OEM) format (Lahiri *et al.*, 1999) was created on top of an object database system, O_2. Ozone is an extension of the standard model for object databases, the Object Data Management Group (ODMG) model (Cattell, 1994) and its query language OQL, to integrate semistructured data with structured data. Ozone supports the Lorel language, but attempts little optimization beyond that offered by the standard OQL

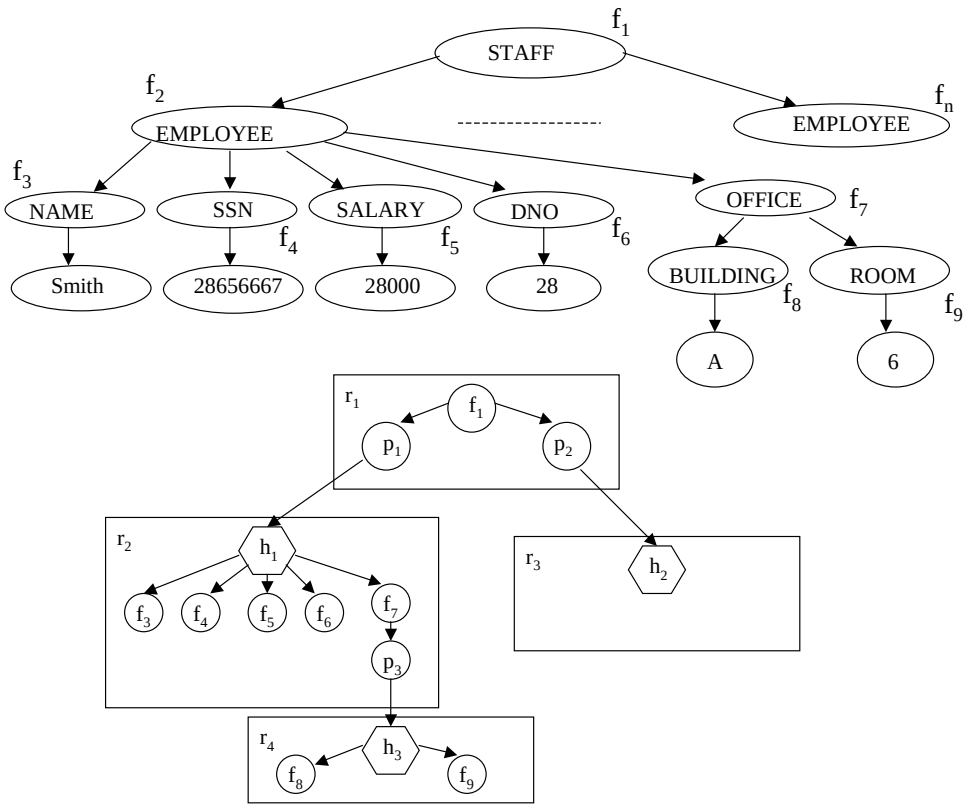

Figure 3. Logical and physical tree nodes for NATIX.

optimizer of O_2. OEM data are stored in O_2 in a simple manner, and Lorel queries are translated into OQL queries. The basic extension of Ozone to the ODMG data model is the definition of a new class type OEM (discussed later). Lore's OEM is a simple, self describing, nested object model that can be considered a labeled directed graph (Goldman *et al.*, 2000) which is similar to the DOM. In the OEM, all entities are objects, either atomic or complex and each object has a unique object identifier (oid). Atomic objects contain base values such as integers, string, *etc.* and complex objects can contain other objects. Objects in the Ozone class OEM are of two categories – OEMcomplex, and

OEMatomic, representing complex and atomic OEM objects respectively. An OEM complex object encapsulates a collection of (label, value) pairs, where label is a string, and value is an OEM object. This class has two sub-classes, OEMcomplexset, and OEMcomplexlist for representing ordered and unordered objects inside the OEM complex object.

Consider the DOM for the XML document (staff.xml). The three OEM complex objects for this document are:

```
OEM_Staff
        (“employees”, OEM(list(Employee)))
OEM_Employee
        (“name”, OEM_string)
        (“ssn”, OEM_integer)
        (“salary”, OEM_integer)
        (“dno”, OEM_integer)
        (“office”, OEM(Office))
OEM_Office
        (“building”, OEM_string)
        (“room”, OEM_integer)
```

For performance reasons, Ozone allows the definition of auxiliary classes. For example, a proxy class, OEM_Employee_ssn can be used to encapsulate the social security number. A query on an auxiliary class object would be faster than a query over the equivalent OEMcomplex object. So, in order to obtain the ssn of all the employees, the system would access OEM_Employee_ssn rather than OEM_Employee. In indexing terminology, this would be termed as a path index (explained later). Efficient retrieval of data is achieved with the help of auxiliary classes.

3.1.3 XSet

XSet stores entire XML documents either in main memory, or on disk. As XSet is developed in Java, XML data are serialized for storage on disk. The authors note that this storage technique will be modified later. In order to achieve efficient retrieval of data, XSet uses a unique method. Each document is assigned a monotonically increasing unique identifier which is also used in paging and logging operations. When the document is loaded into main memory, it is merged into an index (discussed later). XSet provides the user with the facility to explicitly page documents in and out of the memory while keeping the indexing information in the main memory. So, while documents resides on disk, the system still has access to all the information via the index. If a

document is required for a query, it can be retrieved quickly. While it is advantageous to keep the index information on main memory, the memory overhead restricts the number and size of documents that can be used in this database system. Updates to the data are logged to a finite size log buffer which is periodically flushed to the disk, thus updating the documents.

3.1.4 QuiXote

QuiXote has its own query language QNX. The QNX data model used by QuiXote views XML documents as a set of <schema, setOfData> pairs. All documents that conform to a given schema are grouped together and associated with this schema. Since QuiXote captures intra-document referencing (IDREF), data in an XML document are viewed as a graph. Figure 4 shows the partial view of the QNX data model for the staff.xml. The numbers on each node are called element references, or element addresses.

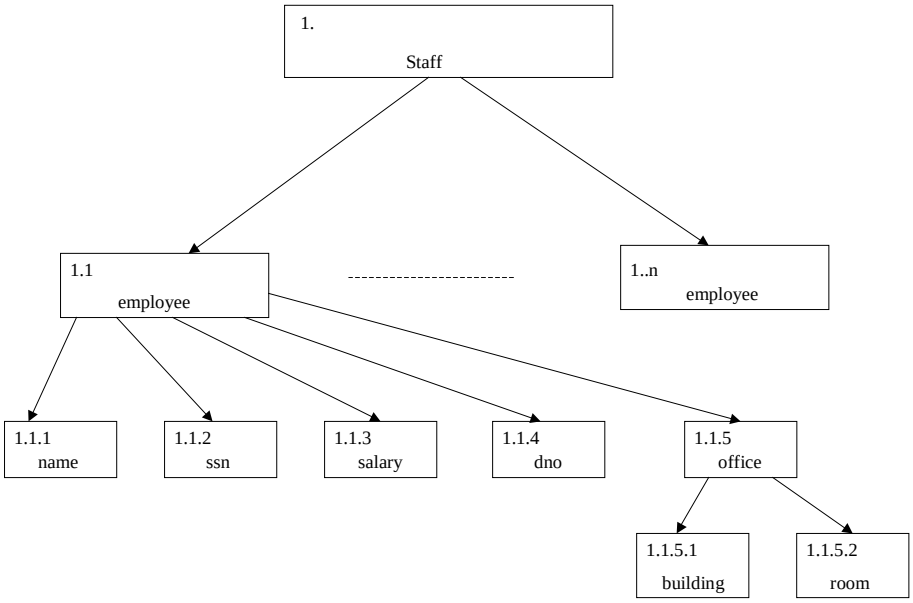

Figure 4. The QNX data model.

Because QuiXote stores each XML document as a tree, parent-child relationships are captured explicitly, but link relations are captured implicitly using link indices. Link indices are simple mapping from ID names to elements. QuiXote uses a binary storage model, Millau, to store the XML tree (Girardot and Sundaresan, 2000). Millau is

designed for efficient encoding and streaming of XML structures. Miallu uses the Wireless Application Protocol (WAP) binary XML format, a compact binary representation of XML, which was designed to reduce the transmission size of XML documents without compromising their functionality or information. Compression is achieved by using tokens for element and attribute names specified in the DTD and then applying standard text compression for attribute and text values. Experiments on Millau show that it can achieve compressions of about 80 percent. In addition, Millau also defines a load model permitting portions of documents to be loaded into memory as and when they are needed.

3.1.5 MMXDB

MMXDB is a main memory database system where all data are loaded into memory prior to query execution. Secondary storage is only used for backup purposes. Like Lore, storage is object oriented but, unlike Lore, an underlying object oriented database is not used. Consider the schema definition statements for staff.xml as they would be entered by the user. The syntax is as defined by the XML Algebra of Fernandez *et al.* (2001).

```
type Staff =
   staff [ Employee{0, *} ]
type Employee =
   employee [
                        @ssn [ Integer ],
                        name [ String ],
                        salary [ Integer ],
                        dno [ Integer ],
                        office [ Office ]
                ]
type Office =
    office [
                        building [ String ],
                         room [ Integer ]
            ]
```

The first statement defines the staff element as a collection of zero or more employee elements. The second statement defines the employee element and all its sub-elements including the office element. The third statement defines the office element which contains the building and room elements. As can be seen from the above statements, each element of the document that has sub-elements is defined separately. In MMXDB, each element that has sub-elements becomes a separate object as shown:

```
staff
        employee        list(staff_employee)
staff_exployee
        name            string
        ssn             integer
        salary          integer
        dno             integer
        office          staff_employee_office
staff_employee_office
        building        string
        room            integer
```

Each of the above objects translates to a Java class (see Appendix A for code). In the class defined for the staff object, the element employee is represented by a List collection object containing references to all staff_employee objects. Each type of object defined has an associated extent that references all objects of that type. Thus, for the objects defined above, we would have staff_extent, staff_employee_extent, and staff_employee_office_extent. The data population statements as defined by the algebra are shown below (Fernandez *et al*., 2001).

```
let staff0 : Staff =
 staff [
  employee [
        @ssn [ 28656667 ],
        name [ “Smith” ],
        salary [ 28000 ],
        dno [ 28 ],
        office [
                building [ “A” ],
                room [ 6 ]
               ]
  ],
  employee [
        @ssn [ 12345678 ],
        name [ “Clark” ],
        salary [ 18000 ],
        dno [ 18 ]
        office [
                building [ “A” ],
                room [ 7 ]
               ]
  ]
 ]
```

Since staff0, as shown above, is an instance of staff, staff_extent stores a reference to the staff0 object with string “staff0” as the key. Similarly, if there was another instance of staff (staff1) defined, it too would also be referenced in the staff_extent. Although, as seen above, the user has not explicitly specified keys for employee and office objects, these are stored in the staff_employee_extent and

staff_employee_office_extent using system generated object references. Storing object references in extents is used for indexing purposes (described later). All extents are loaded into memory during system startup. The object oriented approach to storage was chosen since it allows an intuitive and easy mapping from the user entered schema statements to objects as shown above. It is also easy and efficient to manipulate and store objects using high level object oriented languages such as Java.

3.1.6 Comparison

While Lore, depending on the version, stores data in an object oriented database O_2 or in native storage. XSet and QuiXote use a native storage model. XSet, being a light weight system, stores entire documents in the main memory index. Thus, it relies heavily on its indexing structure for efficient retrieval of individual elements. QuiXote in addition to storing XML documents modeled as trees, uses the Millau compression system for efficient use of space. Thus, QuiXote has the most space efficient storage system. MMXDB modeled the document into object classes, but to store XML documents on disk, object serialization was used.

New mechanisms such as XPointer, XPath, and Xlinks can be used inside XML documents to create inter-document links. However, as these are relatively new terms in the XML vocabulary, none of the systems we studied discussed them. It can be assumed that future storage structures will have to address these issues and provide mechanisms to handle such associations between documents.

3.2 Indexing

3.2.1 Introduction

Most XML databases parse XML data and load it into a tree like structure similar to the DOM. One way to solve queries would be to start traversing the tree from top to bottom, taking all possible paths till we find the information we are looking for. This would be a very inefficient approach. Thus, indices are required to address this issue of path navigation. In fact, three basic issues have to be addressed by indices so as to reduce the search time for a query. These are 1) reduce the search on all similar paths by clustering them together 2) reduce the search on values by clustering them together and

3) assist in solving regular expressions. 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 can be considered to be collection of all elements and/or attributes that can be reached by following the same path. Before discussing indices used in the selected databases, we discuss the idea of a simple path index as explained by Abiteboul *et al.* (1999). For XML data stored in a tree, they propose an index which itself is a tree. Each node in the index is a collection (that my be implemented as a hash table). Each entry in this collection contains a list of references to the corresponding nodes in the data tree. For example, in the index tree of Figure 5, two paths (building and room) can be traversed from node h3. So, node h3 has two entries, one for the collection of references to all building nodes of all offices of all employees, and the other for references to all room nodes. Thus, the index tree has one and only one node/hash table for every sequence of labels leading to a non-leaf node in the data tree. Each entry in the hash table holds a collection of references to all nodes reachable by the same path expression. For example, references to all employee.name nodes, n_1 n_2, n_3, … n_n are associated with key 'name' in hash table h2. So, if the query has to access all names at this level (employee.name), the system traverses the index tree to name and accesses this list of pointers.

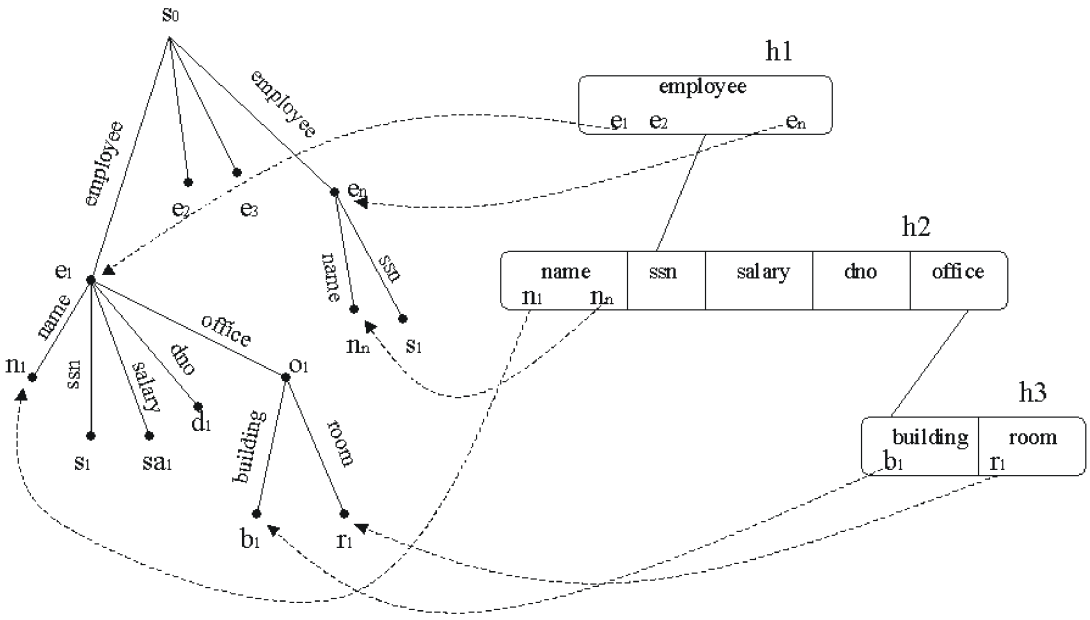

Figure 5. Tree and index for staff.xml.

In order to evaluate Q1, the index is followed to the node containing name, avoiding the traversal of the entire tree. But, because this is a path index and values are not indexed, pointers to all the name nodes have to be traversed to find the “Smith” node. However, this problem can be eliminated by creating an index of all values that are reachable by the same path. It would be pertinent to question the need for both the data tree and the index. However, once the employee has been found using the index, the rest of the information on “Simth” can only be reached using the data tree. Generalized path expressions can also use the index, although not as efficiently. Consider the query (company.employee.*.room). The index is traversed to employee, but then all the entries there (name, ssn, salary, dno, office) have to be searched to find a subset (office in this case) that has room as the next entry. The drawback is that the earlier you encounter a wild card, the earlier you have to consider all the nodes. So, if the first entry in the query’s path was a wild card, it would slow down the evaluation considerably. This issue of wild cards in regular path expressions is addressed by Milo and Suciu (1999) and Ives *et al.* (2000). While issues relating to path expressions are addressed by most of the XML database systems, regular expressions are not typically considered. Regular expressions consist of concatenation, closure, and alteration. An example of a regular expression is employee.name = <Sammy | .mith>, where the user wants to find any employee with the name Sammy or any other five character name ending in mith. Efficient regular expression pattern matching will be important for search engine type applications of XML databases. We now consider indexing issues with respect to the selected XML databases.

3.2.2 Lore

Lore provides a whole suite of indices (value, label, edge, text, and path) each of which are discussed below. Lore has physical operators to represent all indices except the path index which are expressed as a dataguide.

The Vindex or value index indexes all atomic objects (integer, real, string) that have an incoming edge with label l. While the edge can be omitted in the search, in which case, all atomic values are searched, it helps to associate the edge with the value because “a specific desired incoming label is usually known at query processing time, so it is

useful to partition the Vindex by label" (McHugh *et al.* 1998). The value index supports coercion so that comparison between two different atomic types can be achieved. The physical operator, Vindex(l,Op,Value,x) accepts a label l, an operator Op, and a Value and places into x all atomic objects that satisfy the "OpValue" condition and have an incoming edge labeled l.

Because Lore does not support reverse references (from child to parent), a value index would be of little use if one could not traverse from the value leaf up the tree. In order to overcome this problem, Lore offers a label index that retains in it, the association between a child and parent via a label l. The physical operator Lindex(x,l,y) places into x all objects that are parents of y via an edge labeled l.

In addition to the label index, Lore strengthens random access to the tree and easy traversal by providing an edge index or Bindex. This index holds information on all the parent child object pairs that are connected via a specified label, l. The physical operator Bindex(x,l,y) finds all parent child nodes connected by l and places them in x and y respectively.

Lore also provides an information–retrieval style full text index (analogous to regular expression pattern matching) based on the concept of inverted indexes. When a search for a word 'w' is applied, Tindex returns a list of pairs (o,n) which indicates that node o (always a leaf node) contains the word 'w' which is the n^{th} word in its value string. This index can be used to search the database for specific words or groups of words.

The last kind of index, Pindex or path index, uses dataguides. A dataguide is "a dynamic structural summary of all possible paths within the database at any given point in time" (McHugh *et al.,* 1998). Thus, all objects reachable by a specific path can be accessed using the Pindex. A disadvantage of Lore is that the indices may consume enormous amount of memory sometimes, even more than the data set itself (Goldman *et al.,* 2000). Also, every time the database is updated, indices must also be modified. Thus, database updates can be expensive in Lore.

3.2.3 XSet

In XSet, each XML document is parsed and merged into a hierarchical tag index structure. This memory resident index is used to process most of the queries. The tag index for staff.xml is shown in Figure 6. The tag index can be characterized as a dynamic structural summary of the documents and is similar to the path index of Lore.

Element values are stored as sets inside a treap (Seidel and Aragon, 1996) which is a probabilistic self balancing tree structure. For example, consider the first employee of staff from the XML document staff.xml. In this case, the reference to the document containing “Smith” would be inserted into treap T1 with “Smith” as the key, treap T2 with 28656667 as the key, and so on. Nodes in a treap store the value, and also a priority which is used to place the node in the tree hierarchy. Thus, treaps use dual indices to organize keys by value and priority. Since tags are defined uniquely by a combination of context and tag name, they cannot be indexed purely on their tag names. For example, to access ssn, the complete path starting at the root (staff->employee->ssn) has to be specified.

One possible drawback of this system is that it does not index entire elements. For example, in Q1, to obtain all the employee information on “Smith” one would follow, from the treap, the reference to the document containing “Smith”. Then the document would be searched for employee information. Basically, the index only assists in getting to the appropriate document. As compared to Lore, XSet has only one type of index (path index). Although the treaps are indexed by value, the system has to traverse the path index to get to them. Nevertheless, since XSet is intended to support a limited set of queries, this is sufficient for their application (Zhao and Joseph, 2000).

3.2.4 QuiXote

The QuiXote indexer provides for three kinds of indices. These are value index (includes text and attribute indices), structure index, and link index. Separate index structures are maintained for each document type in the repository. The compressed format provided by Millau is used to reduce the storage cost of the indices.

Text indices are maintained for values contained in the text nodes of elements (represented by CDATA or PCDATA in the DTD of the document). The text value index

is a set of (V,S) pairs where V is the value key used for retrieval, and S is a set of element references or its ‘address’ as described in the discussion on storage. These indices are

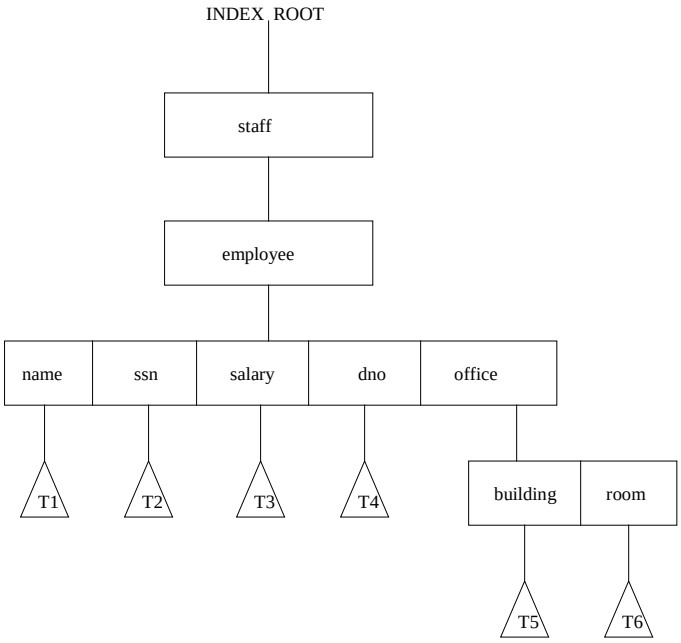

Figure 6. Index structure for XSet.

maintained as a set of (E, {V_1,S_1), (V_2,S_2), ..V_n,S_n)}) pairs, where E is an element tag name, V_i is a text string, and S_i is the set of references to elements in the document that satisfies the key (E,V_i). The text index for the XML document staff.xml is shown in Figure 7. For example, reference to “Smith” that can be used by Q1 will be stored as (Name, {“Smith”, 1.1.1}). Attribute indices are similar to text indices but are used store attribute information. The structure index for QuiXote (Figure 7) is similar to the path indices discussed earlier. The link index is a simple mapping from ID names to elements.

3.2.5 MMXDB

MMXDB provides a path index as shown in Figure 8. References to all element nodes that do not have sub-elements are stored in their own extent and hence automatically indexed. For instance, if a query required a list of all the office buildings of all the employees, the system could easily access all the office objects from the staff_employee_office_extent. Thus, the staff/employee/office path is already indexed.

Similarly, to locate all employees, the system would use the staff_employee_extent. The extent staff_extent which references the root elements may be required for indexing but we have defined it for consistency of the storage model.

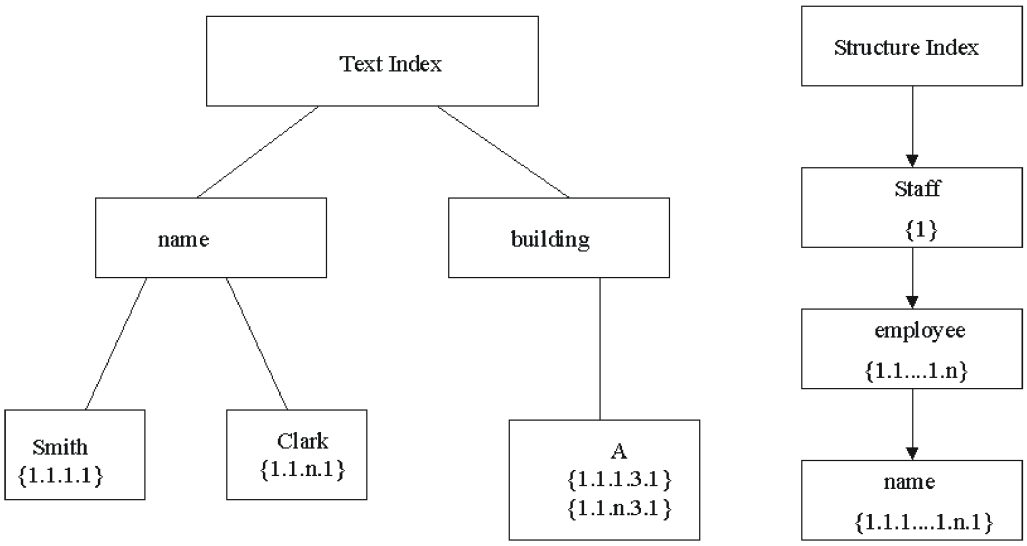

Figure 7. Text and structure indices for QuiXote.

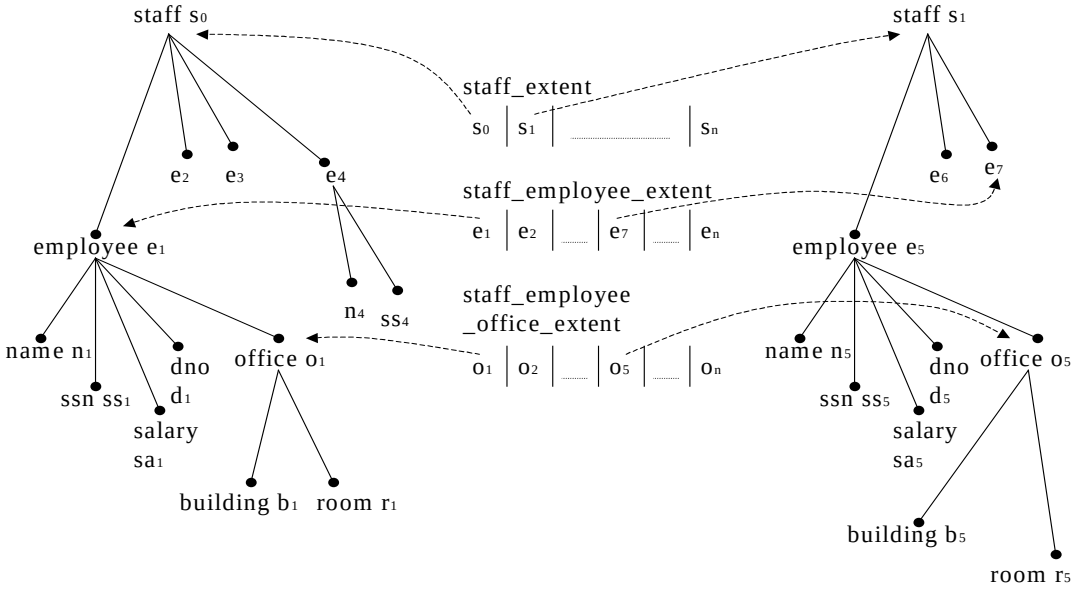

Figure 8. Document trees and extents for MMXDB.

3.2.6 Comparison

Efficient query execution relies heavily on indexing strategies. Of all the XML database systems studied, Lore probably provides the most extensive indexing capabilities (value, label, edge, text, and path indices). This allows for a single query to be executed in different ways using different combinations of indices. A cost model is used to select the best execution strategy and thus every query is executed in an efficient way. The main drawback is that the indices may occupy as much or even more disk space than the data. XSet, on the other hand, does not have as many indices. The XSet index is built such that all the paths in the documents are indexed. Since XSet is a light weight system that supports a minimal set of queries, this indexing method is sufficient for its target applications. QuiXote, like Lore, also maintains value, structure, and link indices. However, because QuiXote uses Millau compression for storage, it is not as space inefficient as Lore. MMXDB, for now, provides only a path index.

3.3 Query Evaluation

3.3.1 Introduction

The query evaluator executes the user query based on the query evaluation tree provided to it by the query planner. The major approaches to evaluating a query are the predicate based approach used in relational databases, and the functional approach (Berri and Milo, 1997).

In the predicate approach, the logical query plan is an optimized algebraic representation of the query. For example, in the relational model, the logical query plan is composed of relational algebra operators such as join, select, *etc.* Physical query plans are then built from logical query plans. These physical operators are usually the implementations of logical operators and various algorithms are defined for these operators (Garcia-Molina *et al.*, 2000). Graefe (1993) discusses in detail the algorithms used to evaluate physical operators for the relational model. The purpose of generating a physical query plan is to take advantage of the storage and indexing mechanisms of the system to efficiently evaluate the queries. The physical tree can be evaluated either using a top-down, bottom-up, or a hybrid approach. The top-down approach is to start at the root of the tree using all possible path expressions, whereas, the bottom-up approach is to

find all objects that satisfy the predicate and then move backwards up to the root. All three approaches are used are used in the Lore system and are explained in detail by McHugh and Widom (1999). The main advantage of the predicate approach is that the logical query plan or tree can be optimized by rearranging the operators. It also allows for features such as short-circuit evaluation. Also the predicate approach is founded on well researched and widely understood concepts. This may be the reason why this approach is used in systems such as Lore and QuiXote.

In the functional approach the problem of evaluating of a query is converted to the problem of evaluating functions. The algebra operators are expressed as functions and the query evaluation is recursive. Although the functional approach, due to its recursive nature, lends itself well to query tree evaluation, it has some drawbacks. The query plans created using this approach, although easy to evaluate, are fairly inflexible as far as re-ordering of certain operators such as joins is concerned. So, creative storage and indexing strategies may be required for optimization. For example, data storage in MMXDB clusters documents that have the same path, thus eliminating the need to scan the entire tree for a particular element. We now discuss the evaluation techniques for each of these database systems in depth.

3.3.2 Lore

Query evaluation in Lore is based on the predicate approach. Every query is transformed into a logical query plan using logical operators such as Select, Project, Discover, Name, *etc*. These can also be considered algebra operators. Each logical query plan can give rise to several physical query plans and a cost based approach is used to select the best physical plan among them. Depending on the type of the physical query plan chosen by the cost based optimizer, either a top down, bottom up, or hybrid query evaluation strategy is used by the system. The logical query plan and one of the several possible physical query plans for query Q1 are shown in Figure 9.

Discover and Chain are logical operators used for path expressions. Each simple path expression in the query is represented as a Discover node, multiple path expressions are grouped together as a tree of Discover nodes connected via Chain nodes. Places where independent components meet are called rotation points (represented by the Glue

node). During the creation of physical query plans the order between independent components can be rotated to get different physical query plans. The CreateTemp and Project nodes at the top of the plan gather the variables that satisfy the evaluations and return the appropriate objects to the user. Although the authors have not given examples of queries such as Q2, involving two or more documents, we assume that it would involve use of the Glue, Set, and CreateSet logical operators. The Set operator performs set operations such as union, intersect, *etc*. using two sets of CreateSet structures passed up from the children nodes.

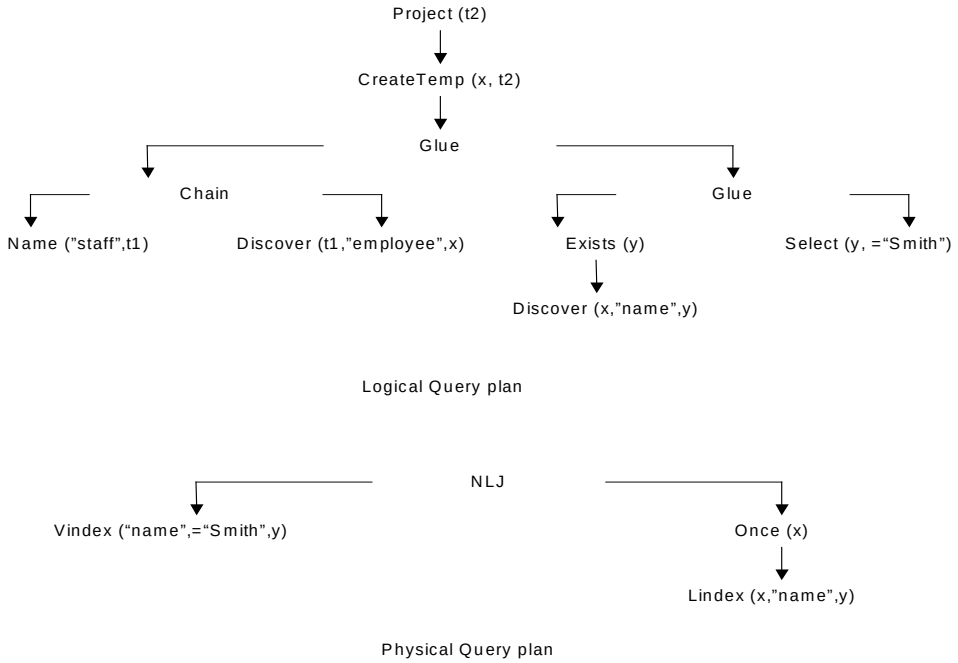

Figure 9. Logical and physical query plans for Lore.

The physical query plan uses physical operators such as NLJ (nested-loop join), Vindex, Lindex, and Once to execute the query. In Q1, the Lindex operator places in x all objects that are parents of y via the edge 'name'. The Once operator allows the name to be passed to the parent only if has not been seen before. The Vindex operator uses the value index to find all objects that satisfy the condition name = 'Smith'. The NLJ operator, passes bound variables from the left subplan to the right subplan. The details for these operators are provided in McHugh and Widom (1998).

3.3.3 XSet

XSet queries are themselves well-formed XML documents, with optional embedded query instructions for the query processor. In query processing, collections of document references that match each search constraint undergo global intersection to return the result set. The query Q1 is written as:

```
<staff>
    <employee>
        <name>Smith</name>
    </employee>
</staff>
```

Tags that are not explicitly stated in the XSet query are assumed to be wildcards that can match any XML tag value or subtree. No examples have been provided of queries similar to Q2 which query across two documents. XSet may not support such queries since the authors state that their query model can be characterized as a very limited subset of XQL (Robbie *et al*., 1998). XQL is an XML query language with syntax similar to directory like notation for path expressions.

3.3.4 QuiXote

The QuiXote system defines an XML query language QNX which is used by the user to query the data. Query processing is carried out using two parts. The first part, the preprocessor, precompiles structural relationship information and generates indices based on the schema and data. The second part, the query processor, processes the user query.

The preprocessor consists of the schema extractor which extracts the schema for XML documents that do not have a predefined schema; the relation set generator and the indexer. The indexer builds value, structure, and link indices, and the relation set generator computes relationship sets between elements and attributes from the schema. Some example relationship sets of an element A are child, parent, ancestor, reachability, *etc.* The child set is the set of possible child elements of A, parent set is set of its possible parent elements (needed when a particular element appears as a sub-element of two or more elements, *e.g.* name of employee and name of department) and the reachability set

is the set of all descendents of A and the levels at which that descendant can be present. For example, the reachability set for the element staff is:

{(employee, {1}), (name, {2}), (ssn, {2}), (salary, {2}), (dno, {2}), (office, {2}), (building, {3}), (room, {3})}

The query Q1 would be expressed in QNX as shown below:

```
<Query qnx:“QUERY” = “ EmployeeSmith”
        <FROM Source = “staff.xml”/>
        <qnx:PATTERN>
                <employee>
                   <name>
                      <qnx:PCDATA qnx:OPERATOR = “eq”  qnx:VALMATCH = “Smith” />
                   </name>
                </employee>
        </qnx:PATTERN>
</Query>
```

The query processor, consists of the document filter, the query optimizer, the query executor, and the schema generator. The document filter filters out documents that will produce an empty result set, *i.e.*, documents that do not contain the information being searched. For query Q1, staff.xml is not rejected because it contains information relevant to the query and so this document is sent to the next stage, the query optimizer. The query optimizer performs two tasks - strength reduction of expensive query constructs (replace complex query constructs by simpler ones), and choosing an optimal query plan. Relationship sets are used for strength reduction. For query Q1, a strength reduction can be performed for employee. The query asks to select all employee from the document since the employee element can be present at any depth from the document root element – staff. From the reachability sets, we know that employee elements can be present only at a height 1 from the document root element, so the processor needs to check only the child elements of staff for employees. After this, candidate Query Execution Plans (QEPs) are specified for the “reduced” query and a cost model is used to estimate their cost. A cost effective QEP is selected and passed to the query executor for execution. Two example QEPs are shown in Figure 10.

The QEP specifies a tree of operators. The operators used in Figure 10 are the root operator, the descendent operator, the ancestor operator, the text operator, and the index Scan operator. The descendent operator is denoted by Des(E,N,L), where E is the element name, L is the depth, and N is the count used to indicate how many of the candidate elements for this operator should have element name E. The text operator is denoted as

Text(Op, S) where Op is an operator, and S is a string or variable reference. The index scan operator is denoted by Index(F, S, K) where F is the index file name, S is the kind of index, and K is the key. The query executor executes the QEP obtained from the optimizer. Query execution starts from the root node. Finally, the schema generator generates the DTD for the result set produced. In this system also the authors have not provided any examples of queries similar to Q2 which query across two documents. We assume that it could be done using the join operator which combines results obtained from its child operators based on a condition.

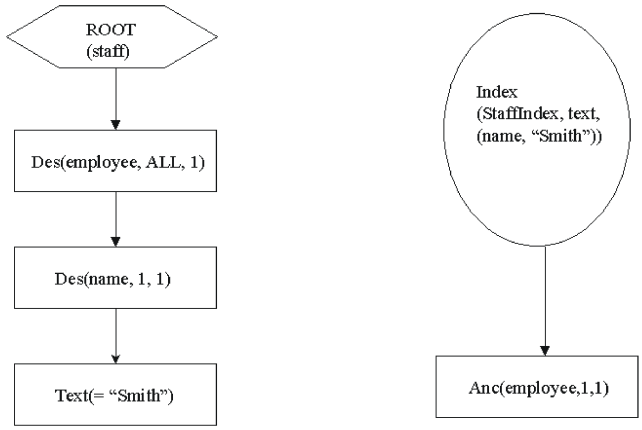

Figure 10. Two alternate query evaluation plans (QEPs) for Quixote.

3.3.5 MMXDB

Since the algebra in which MMXDB queries are expressed in is a functional language, we use recursion to evaluate queries. This is because functional languages lend themselves well to recursive evaluation. The query Q1 which finds all employees whose name is Smith would be represented in our algebra as follows:

```
for e in employee(staff0) do
  for n in name(e) do
        if data(n) = "Smith" then e
```

The algebraic constructs used in this query are:

```
Exp ::=  for Var in Exp do Exp        ForExp
Exp ::=  if Exp then Exp              IfExp
Exp ::= Exp BinaryOp Exp              BooleanExp
Exp ::=  FuncName(Exp)                FuncNameExp
Exp ::= Var                           VarExp
FuncName ::= data | Var
```

The query plan for the above algebraic query expression, generated by the query planner, is as shown in Figure 11. For details on how this plan is generated refer to Chinwala and Miller (2001). The query plan or tree illustrates the recursive nature of the query expression. The topmost expression is the ForExp which when evaluated will give the result of the query. However, the ForExp contains other expressions which in turn have other expressions. Thus, we cannot get the result of the ForExp till we have evaluated all the other expressions. We defined an evaluate function for each type of node in the tree. For instance, the evaluate function for the nodes of type ForExp (for Var in Exp1 do Exp2) is defined as follows:

```
final_result = { }
    temp_result_a = Exp1.evaluate()
    for each entry e in temp_result
    begin
        Var = e
         temp_result_b = Exp2.evaluate()      // here Exp2 would use the latest value of Var
         add temp_result_b to final_result
    end
    return final_result
```

All evaluate functions return a list of values preceded by the type of the result. Once all the evaluate functions are defined for the various types of nodes, evaluating the query is a matter of calling the evaluate function of the root node. In our example, the root node is the ForExp node. The evaluate function of this node iterates over all employee objects in staff0 and passes them to the evaluate function of its child ForExp node. This node, in turn, iterates over the name of employee (since name is not a collection type, only one iteration is performed) and passes it to its child IfExp node. The evaluate function of IfExp node returns the list of employee objects where the name is

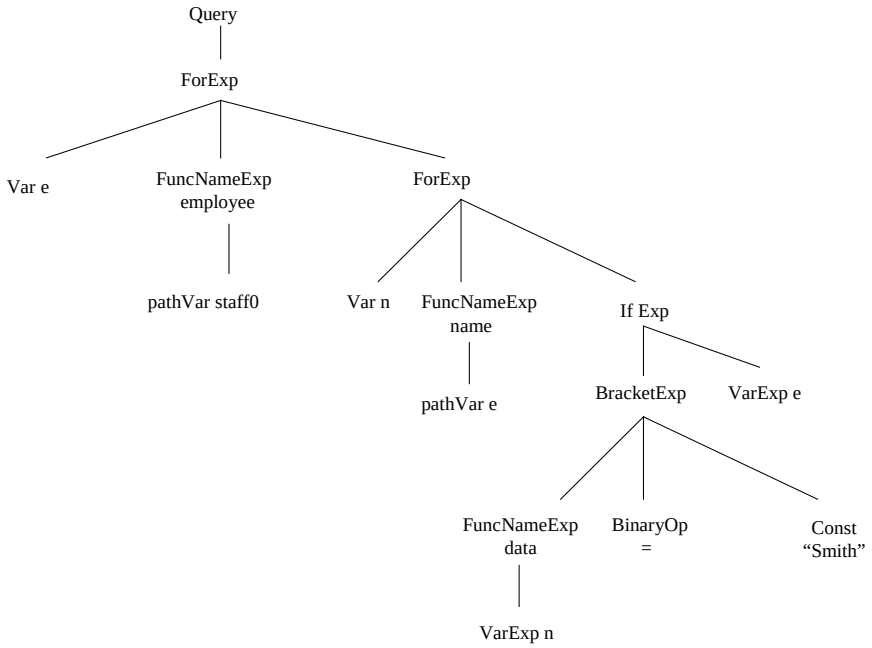

Figure 11. Query plan generated by MMXDB.

‘Smith’. This list of objects is prefixed by string “employee” that denotes the type of objects in the list. This list is passed up the tree and is eventually returned as the result of the query. Inherent path indices are used for efficient access because objects accessible by the same path index are stored together in a collection. For Q1, we would follow the staff_employee path to reach all the employees. The query Q2 would be expressed in our system as follows:

```
for e in employee(staff) do
    for d in department(departments) do
        If ( (for s in ssn(e) do data(s) = for m in mgrssn(d) do data(m)) and
            (for n in dname(d) do data(n) = “MIS”) )  then  name(e)
```

As the above query, which joins employee and department based on ssn, illustrates that the join operator in our system is expressed as nested for loops. So, for each employee, the query iterates over all department elements to find the match on ssn.

3.3.6 Comparison

Lore uses a predicate and cost based query evaluation technique. Various physical query plans are generated for a logical query plan and a cost model is used to select a cost effective plan. A bottom-up, top-down, or hybrid strategy is used to navigate and evaluate the query plan. As XSet is a lightweight system and does not provide for sophisticated queries, the authors do not discuss any query plans. Documents are returned for each individual search criterion and their intersection gives the final result. QuiXote query evaluation is similar to Lore as they also generate several plans and then use a cost model to select a cost effective plan. The authors state that at present, QuiXote lacks a good cost model for choosing between different QEPs. MMXDB generates a single query plan that is recursively evaluated.

4. Transactions, Snapshots, and Security

While relational and object-relational databases automatically provide transaction support and security, not many native XML databases discuss these issues. This may be because native XML databases are still in the nascent stages of development and such issues will be addressed later. Of all the database systems discussed, XSet provides for persistence and failure recovery using snapshots. This system uses fuzzy checkpointing for snapshot creation and recovery. Although the authors claim that the native XML version of Lore is a multi-user system, details of transaction support are not provided.

5. Conclusions and Future Work

This paper addresses important research issues of storage, indexing, and query evaluation that should be considered in developing a native XML database engine. Three XML database systems, Lore, XSet, and QuiXote are used to illustrate the approaches taken to address these issues. The storage, indexing, and evaluation models for each one of these systems was outlined and their salient features compared. In addition, MMXDB, an XML system developed by us using the AT&T algebra selected by the W3C as the proposed standard, was developed and discussed.

XML data can be stored in relational, object oriented, or native database systems. However, as we have seen, native storage of XML data is the most efficient because data

do not have to be mapped to some other format, eliminating a layer between logical and physical storage. This provides for quick retrieval of data because data are clustered together based on the XML schema and not the underlying storage schema. Although XML data and indices may occupy enormous amounts of space, QuiXote provides an elegant solution to this problem. It compresses (Millau) the data and indices for storage. Query evaluation can be carried out using the predicate model (Lore) or the functional model (MMXDB). While the functional model may be intuitively more suited for evaluating data stored in tree like structures, the predicate approach would probably be a better choice. This is primarily because while the predicate approach has been well researched and studied in context of relational databases, the functional approach has not been widely used in database engines.

This study does not imply that native XML databases will become an instant standard. Because companies have invested a considerable amount of time, effort, and money into existing relational database systems, the status quo of extending relational databases for semi-structured data is unlikely to change overnight. However, light weight applications where information is stored using XML could greatly benefit from the creation of XML databases such as XSet. Another important application of XML databases is the allied field of text based searches. Such searches, although not in XML, are continuously running behind today's search engines. The issues is creating such an indexing and retrieval mechanism are addressed by Shin (2001).

Several avenues exist for future work on the MMXDB database systems developed by us. Prominent among them are the development of a more sophisticated storage system, addition of an index structure to index the values of elements, and the development of an evaluation system that uses a physical query plan with physical operators. Currently, we use java serialization to persistently store objects. However, MMXDB should be modified so that information is efficiently mapped from the data structures in main-memory to disk. Thus, although MMXDB has scope for several improvements, it was useful in demonstrating the problems and opportunities in developing a main memory native XML database system. Developing the system was particularly insightful when it came to understanding the issues related to storing and indexing XML data.

BIBLIOGRAPHY

Abiteboul, S., D. Quass, J. McHugh, J. Widom, and J. Weiner. 1997. The Lorel Query Language for Semistructured Data. Journal of Digital Libraries, 1(1):68-88.

Abiteboul, S., P. Buneman, and D. Suciu. 1999. Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaugmann.

Berri, C. and T. Milo. 1996. Comparison of Functional and Predicative Query Paradigms. Journal of Computer and System Sciences. 54:3-33.

Cattell, R.R.G. 1994. The Object Database Standard: ODMG-93. Morgan Kaufmann, San Francisco, California.

Chinwala, M.G., J.Miller. 2001. Algebraic Languages for XML databases. Submitted to Information Systems, August 2001.

Chinwala, M.G., R. Malhotra, J. Miller. 2001. Progress Towards Standards for XML Databases, Proceedings of the 39th Annual ACM Southeast Conference, pp. 277-284.

Codd, E.F. 1970. A Relational Model of Data for Large Shared Data Banks. CACM 13(6):377-387.

Deutsch, A., M. Fernandez, and D. Suciu. 1999 Storing Semistructured Data with STORED, SIGMOD Conference, Philadelphia, Pennsylvania, June 1-3, 1999, pp. 431-442.

Fernandez, M., W-C.. Tan, and D. Suciu. 2000. SilkRoute: Trading between Relations and XML. 9 th Int. World Wide Web Conf. (WWW), Amsterdam, May, 2000

Fernandez, M., J. Simeon, and P.Wadler. 2001. A semi-monad for semi-structured data. International Conference on Database Theory, London, January 2001.

Garcia-Molina, H., J.D. Ullman, and J. Widom. 2000. Database System Implementation. Prentice-Hall, New Jersey.

Girardot, M. and N. Sundaresan. 2000. Millau: an encoding format for efficient representation and exchange of XML documents over the WWW, 9th International World Wide Web Conference, Amsterdam, Netherlands, May 2000.

Goldman, R., J. McHugh, and J. Widom. 2000. "Lore: A database management system for XML," Dr. Dobb's Journal. 25(4):76-80.

Graefe, G. 1993. Query evaluation techniques for large databases. ACM Computing Surveys, 25(2):73-170.

Ives, Z. G., A. Y. Levy, D. S. Weld. 2000. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical Report UW-CSE-2000-05-02, University of Washington.

Kanne, C.-C. and G. Moerkotte. 2000. Efficient Storage of XML Data. Proc. Of the 16 th Int. Conf. On Data Engineering (ICDE), San Diego, March, 2000.

Klettke, M. and H. Meyer. 1999. Managing XML documents in object-relational databases. Rostocker Informatik Fachberichte, 24, 1999

Lahiri, T., S. Abiteboul, and J. Widom. 1999. "Ozone: Integrating structured and semistructured data," Proceedings of the Seventh International Conference on Database Programming Languages, Kinloch Rannoch, Scotland, September 1999.

McHugh, J., S. Abiteboul, R. Goldman, D. Quass, and J. Widom. 1997. Lore: A Database Management System for Semistructured Data. SIGMOD Recor., 26(3):54-66.

McHugh, J. and J. Widom. 1998. Query optimization for semistructured data. Technical report, Stanford University Database Group, August 1998. Document is available as http://www-db.stanford.edu/pub/papers/qo.ps.

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

McHugh, J. and J. Widom. 1999. Query Optimization for XML. In Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, 315-326.

Mani, M. and N. Sundaresan. 2000. Query Processing Using QuiXote, Murali Mani, Neel Sundaresan, IBM Research TRRC 21680 Log 97690 March 2000.

Milo, T. and D. Suciu. 1999. Index Structures for Path Expressions, 7th International ICDT Conference, Jerusalem, Israel, Jan 10 - 12, 1999, pp. 277-295.

Miller, J.A., and S. Sheth.2000. Querying XML Documents, IEEE Potentials (IEEE-STM), Vol. 19, No. 1 (February/March 2000) pp. 24-26. IEEE Press.

Robbie, J., J. Lapp, and D. Schach. 1998. XML Query Language (XQL). In QL '98 – The Query Languages Workshop. Available at http://www.w3c.org/TandS/QL/pp/xql.html.
Seidel, R., and C.R. Aragon. 1996. Randomized search trees. Algorihmica 16:464-497.

Shanmugasundaram, J., K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities, VLDB Conference, September 1999.

Sheth, S.C. 1999. QT4XML: A Query Tool for XML Documents and Databases, Masters Thesis, University of Georgia, July 1999.

Shimura, T., M.Yoshikawa, and S.Uemura. 1999. Storage and retrieval of xml documents using object-relational databases. Proc. of DEXA, Florence, Italy. Lecture Notes in Computer Science, 1677:206--217.

Shin, D. 2001. XML Indexing and Retrieval with a Hybrid Storage Model. Knowledge and Information Systems, 3:252-261.

Tian, F., D. DeWitt, J. Chen, and C. Zhang. 2000. The Design and Performance Evaluation of Alternative XML Storage Strategies. Technical report, CS Dept, University of Wisconsin. Available at http://www.cs.wisc.edu/niagara/papers/vldb00XML.pdf.

Zhao, B.Y. and A. Joseph. 2000. XSet: A Lightweight Database for Internet Applications. Submitted for publication (updated version of MS thesis), May 2000.

APPENDIX A

Generic Java

GJ (Generic Java) is an extension of the Java programming language that supports generic types. It is freely downloadable from:

http://www.cs.bell-labs.com/who/wadler/pizza/gj/

The salient features of GJ are:

- Support for generics. Many data types are generic over some other data type, and this is especially common for reusable libraries such as collection classes. GJ supports the use of such types, for instance allowing one to write the GJ type Vector<String> as opposed to the Java type Vector. With GJ, fewer casts are required, and the compiler catches more errors.
- Superset of the Java programming language. Every Java source program is still legal and retains the same meaning in GJ. The GJ compiler can be used as a Java compiler.
- Compiles into the Java Virtual Machine. GJ compiles into JVM code, so GJ programs run on any Java platform, including Java compliant browsers. Class files produced by the GJ compiler can be freely mixed with those produced by other Java compilers.
- Compatible with existing libraries. One can call any Java library function from GJ, and any GJ library function from Java. Further, where it is sensible, one can assign GJ types to existing Java libraries. For instance, the GJ type Vector<String> is implemented by the Java library type Vector.
- Efficient translation. GJ is translated by erasure: no information about type parameters is maintained at run-time. This means GJ code is pretty much identical to Java code for the same purpose, and equally efficient.

In MMXDB, we used generic java to define object classes for our storage model. Consider the following object class definitions:

```
staff
        employee        list(staff_employee)
staff_exployee
        name            string
        ssn             integer
        salary          integer
        dno             integer
        office          staff_employee_office
staff_employee_office
        building        string
        room            integer
```

The generic java code for the class ‘staff’ is as shown:

```
import java.lang.*;
import java.util.*;

class Staff
{
    public ArrayList <Staff_Employee> employee = new ArrayList<Staff_Employee>();
}
```

Similarly, the code for ‘staff_employee’ is:

```
import java.lang.*;
import java.util.*;

class Staff_Employee
{
        public String name;
        public Integer ssn;
        public Integer salary;
        public Integer dno;
        public Staff_Employee_Office office;
}
```

As seen above, generic java allows us to specify the type of ArrayList employee as Staff_Employee, and the type of variable office as Staff_Employee_Office. If we had used java, both of these would have been of type Object. Thus, generic java allows us to clearly translate object definitions to object classes without losing the type information.