

RC 21680 (Log 97690) 03/03/00
Computer Science

IBM Research Report

Query Processing Using *QuiXote*

Murali Mani
Department of Computer Science
University of California, Los Angeles
mani@cs.ucla.edu

Neel Sundaresan
IBM Research Division
Almaden Research Center
neel@almaden.ibm.com

Limited Distribution Notice

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Some reports are available at <http://domino.watson.ibm.com/library/CyberDig.nsf/home>. Copies may be requested from IBM T.J. Watson Research Center, 16-220, P.O. Box 218, Yorktown Heights, NY 10598 or send email to reports@us.ibm.com.

IBM Research Division
Almaden - Austin - Beijing - Haifa - T.J. Watson - Tokyo - Zurich

Query Processing Using QuiXote

Murali Mani
Dept. of Computer Science
University of California, Los Angeles
mani@cs.ucla.edu

Neel Sundaresan
IBM Almaden Research Center
San Jose, CA
neel@almaden.ibm.com

March 3, 2000

Abstract

With the constant growth in web data and applications, providing a query processing system for this data becomes important. XML (eXtensible Markup Language) promises to revolutionize the web as a common interchange language for data between applications. In this paper we describe a query processing system for XML data called *QuiXote*. The key feature of *QuiXote* is the extensive use of schema (Document Type Definition) for query processing and optimization. For XML documents that do not specify an explicit schema, a schema can always be discovered in a cost-effective manner. *QuiXote* uses structural relationship sets computed from the schema and various structure and value (content) indices in order to decrease the query execution time. The *QuiXote* system defines an XML query language called QNX. QNX is an instance of XML. The *QuiXote* system is a two part querying system - the first part precompiles structural relationship information, and produces indices based upon the schema and data; the second part processes the user query and includes a query plan generator with an optimizer, query executor, and a result schema generator. We describe the *QuiXote* system goals, architecture, and give some experimental performance results.

1 Introduction

The internet is becoming the prime medium for commerce applications and data exchange. Also, XML [31] is poised to become the de facto language for representing internet data. XML, unlike HTML, allows extensible tags that can be used to specify structure. The XML specification requires that all documents are *well-formed* (elements are properly nested). An XML document can optionally be *valid*, that is, it can specify a schema called the DTD (Document Type Definition) [8]. The DTD specifies the structure of the XML document - the elements that can be present, their nesting, and the possible attributes for an element. However, the DTD does not allow data type specification.

Assuming the growing use and availability of XML documents on the web, a general issue is the question of how to efficiently store, index and query XML data. An XML repository might consist of documents that all adhere to one common DTD or documents that use different DTDs. Query systems for XML have to allow to query on different aspects of XML documents: text-based, structure-based and schema-based. Experiments show that query processing time can be significantly reduced through schema-based and index-based optimization techniques. For several queries, we were able to achieve orders of magnitude performance improvement. But XML specification [31] requires that XML documents are well-formed (i.e. have a proper nested structure) but they do not need to be valid (i.e. always have an associated schema with them or that they can be validated against an associated schema). In such cases, schema-based optimization may use automatic schema discovery tools [21, ?, ?, ?, ?] to discover the schema and then use it for optimization.

QuiXote emphasizes on an efficient query processing system for querying XML repositories. It consists of two main modules, the *preprocessor* and the *query processor*. The preprocessor has the task of extracting the schema for documents that do not have a pre-defined schema. Furthermore, to provide for more efficient XML query optimization, *QuiXote*'s preprocessor computes *structural relationship sets* from the schema and *index structures* from the data. *QuiXote* maintains *structure indices* which are based on tree structure patterns, and are used for structure queries, and *value indices* which are used for attribute as well as text value queries. The indices are stored using a binary compression storage model for XML called *Millau* [11]. We are using *Millau* since it provides efficient schema-driven storage algorithms that retain structure. It also defines its own access mechanism for caching and prefetching portions of the document which are suitable for tree traversal operations.

The *QuiXote* query language, called QNX, is a general DTD based query language, which operates on an underlying DTD algebra. The QNX query language is an XML instance, that is, a query in QNX has an XML syntax. We can compute a DTD for the query language¹ and use it for validating user queries (check for a possible empty result set) during query formulation itself. Also using tree structures for queries, as opposed to path expressions makes QNX easier to use as observed from our experience. (We have used QNX as a query language for mining applications on XML repositories.)

The *QuiXote* query processor processes queries in the QNX query language. The query processor has stages for filtering the input documents to eliminate documents that will result in empty result sets. This is done using the structural relationship sets computed during the pre-processing stage. The relationship sets are also used to eliminate redundant (always true) conditions specified in the query² and to simplify expensive query constructs. The query processor then generates an "optimal" Query Execution Plan (QEP), which specifies any indices that should be used. The QEP is then executed using tree traversal operators (based on the DOM [7] specification) defined by *QuiXote*.

1.1 Related Work

Significant research has been done on querying semistructured data³. The first step in building a query processing system is to define an appropriate data model that will capture the necessary information from the data. Various data models have been proposed for XML. One of them is the semistructured data model which describes a directed labeled graph model for an XML document. There are two main deficiencies in using this data model for XML. (1) Order among children of a node is not defined in this model. But order is important to support order queries⁴ and also to render XML documents. Modifications of the semistructured data model that support order specification are used by Lore [17, ?] and XML-QL [5]. (2) The semistructured data model is used for data which has no schema. Therefore this data model becomes less useful when a schema is defined [29].

Using relational data model for XML is studied in [28, ?]. In [28], the DTD for the XML document is used for constructing the relational schema. Relational schema can capture only a subset of the DTD semantics. The use of the relational data model therefore typically requires DTD simplification. Capturing only a portion of the DTD semantics typically will not result in incorrect query processing. But if updates are to be supported for XML, then we would require that the data model capture the entire DTD semantics. *QuiXote* uses the QNX data model that can capture the entire DTD semantics, and maintain order information among the child nodes of an element.

A number of query languages have been proposed for XML. These languages can be classified into two broad categories - those that are based on path expressions (Lore [2] and XQL [24, ?]) and those that use tree structures (XML-QL [5]). Lore relies on a OQL type syntax for the query

¹A DTD for the query language can be obtained as a function of the DTDs of the subject documents.

²Note that it is the structural query conditions that are eliminated.

³Semistructured data is data without an explicit schema. Here the data itself is considered as specifying the schema. An XML document without a DTD can be thought of as semistructured data.

⁴For example, *select the first author of all books* is an order query

language, whereas XQL combines the SELECT, FROM and WHERE clauses into a single clause. XML-QL, on the other hand, uses tree structures instead of path expressions. It is not difficult to observe that a tree structure can be decomposed into multiple path expressions. Both Lorel and XQL support multiple path expressions and therefore all the three query languages effectively have the same expressive power. From our experience in using query languages for applications, we observe that a query language with a tree structure is typically more suited for an XML query language.⁵

Query optimization for XML, like any other database system, relies on schema based optimization as well as indexing. The systems which use the semistructured data model rely on structural summaries computed from the data for optimization (as there is no notion of schema). Lore calls these structural summaries *dataguides* [12, ?]. These structural summaries are usually expensive to compute and could be as large as the database itself. Less expensive techniques for computing structural summaries are discussed in [18]. Lore uses these structural summaries to simplify complex query conditions [20]. Using the DTD for schema-based optimization is discussed in [3]. Here the authors describe how various rules that can be used in query optimization can be computed from the schema. *QuiXote* uses the schema to compute structural relationship sets. The relationship sets are then used extensively in query optimization - detect documents that will produce an empty result set for a query, remove redundant conditions in a query and simplify complex conditions in the query.

Indexing is crucial for any efficient query optimization. Most of the existing index schemes for structure queries define indices for path expressions. Lore defines simple indices - value index, link index (for reaching a parent) and edge index (for querying on edge labels⁶). Lore also uses its dataguides as path indices. Maintaining structure indices for path expressions as nested index and path index are described in [4]. *QuiXote* maintains structure indices (for tree structure patterns), value indices (for document content) and link indices (for querying links).

1.2 Organization of the Paper

The rest of this paper is organized as follows. In the following section we give an overview of XML and DTD. After this we describe the data model used by *QuiXote* (which we call the QNX data model), and the *Millau* storage model. Section 4 gives an overview of the QNX query language. The next section describes the *QuiXote* system architecture. We describe the various stages in preprocessing⁷, the different relationship sets computed, and how the various index structures are maintained. We then describe the various stages in the *QuiXote* query processor⁸ - filtering out documents that will result in an empty result set, simplifying complex conditions specified in the query, generating an optimal QEP, and executing this QEP. In Section 6 we describe the experiments we performed to evaluate *QuiXote*. This includes a basic evaluation of our system - studying scalability of the *Millau* storage model and comparing nested object navigation in *QuiXote* as opposed to a relational database. We then measure the performance gain in using the various optimization techniques - computing the relationship sets and maintaining index structures.

2 Overview of XML and DTD

XML is used to represent both hierarchical structure and data. Structure is represented using nested *elements* and data is represented using *text nodes* and *attribute values*. An XML document can optionally have a schema (DTD) which describes its structure. Example 1 shows a sample XML

⁵Our aim was not to design a new query language. But the incompleteness of existing query languages and a desire to get a first hand feeling of XML query language requirements prompted us to define QNX as our query language.

⁶In XML, edge labels are element tags and also attribute names used for specifying links.

⁷Preprocessing includes all stages from the time a new document enters the repository till it is available for querying

⁸Query processing includes all stages from the time the user asks a query till the result is sent back to the user

document and a DTD for that document. This document represents a small library database consisting of two books and one monograph.

DTD

```
<!DOCTYPE library [  
  <!ELEMENT library (book|monograph)*>  
  <!ELEMENT book (title, author*, publisher?)>  
  <!ELEMENT title (#PCDATA)>  
  <!ELEMENT author EMPTY>  
  <!ATTLIST author firstName CDATA #REQUIRED    middleInitial CDATA #IMPLIED  
    lastName CDATA #REQUIRED >  
  <!ELEMENT publisher (#PCDATA)>  
  <!ELEMENT monograph (title, editor+, publisher?)>  
  <!ELEMENT editor (monograph*)>  
  <!ATTLIST editor firstName CDATA #REQUIRED    middleInitial CDATA #IMPLIED  
    lastName CDATA #REQUIRED >  
]>
```

XML Document

```
<library>  
  <book>  
    <title>The XML Handbook</title>  
    <author firstName='Charles' middleInitial='F' lastName='Goldfarb' />  
    <author firstName='Paul' lastName='Prescod' />  
    <publisher>Prentice Hall</publisher>  
  </book>  
  <book>  
    <title>XML Complete</title>  
    <author firstName='Steven' lastName='Holzner' />  
    <publisher>McGraw-Hill</publisher>  
  </book>  
  <monograph>  
    <title>Active rules in database systems</title>  
    <editor firstName='Norman' middleInitial='W' lastName='Paton' />  
    <publisher>New York : Springer</publisher>  
  </monograph>  
</library>
```

Example 1 A DTD and a sample XML Document (library.xml) conforming to the DTD

2.1 DTD as the XML schema Language

The DTD language is commonly used as the schema language for XML. There are two key features which make DTD different from traditional schemas - 1) DTD allows no data type specification for text fields. 2) It allows rich structure specification through nested *element declarations* (An element declaration is the declaration of the *content model* or structure of an element). A regular expression language with operators {*, +, ?, ,, |} is used to describe the content model in an element declaration

(See Example 1).

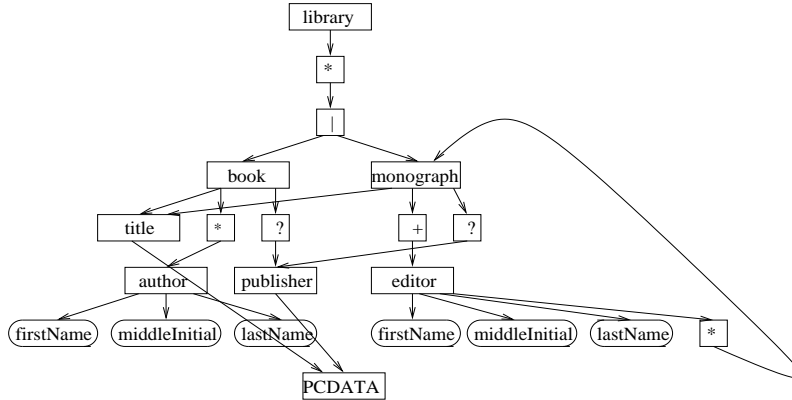


Figure 1: Graphical representation of the DTD of Example 1. Elements, attributes and the regular expression operators used to represent the content model form nodes in the graph. The `monograph-editor-monograph` cycle in the graph is because of the recursive element declaration.

The DTD is best viewed as a directed graph, rooted at an element called the DOCTYPE⁹ declaration. The directed graph view of the DTD in Example 1 is shown in Figure 1. The entire DTD semantics cannot be captured by relational schemas. Therefore DTD “simplification” is necessary for XML-relational conversion. In [28], parent-child relationships are captured using $\{*, ?, ONCE\}$.¹⁰

2.2 Information in an XML Document

An XML document represents two kinds of information - structural information through nested elements and textual information through attribute and text values. The structure in an XML document is typically viewed as a graph, rooted at the document root element. See Figure 2.¹¹ We use an address¹² to identify each element [16]. Given such an address, the corresponding element in the document can be reached in $O(h)$ steps where h is the height of the document tree.

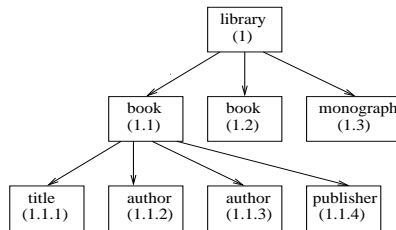


Figure 2: Graphical representation of a portion of the XML document in Example 1. The value associated with an element (address) uniquely identifies the position of the element in the document.

⁹This is different from how a DTD is interpreted in [28]. They consider the DTD as *not* having a root element as they do not consider the DOCTYPE declaration as part of the DTD.

¹⁰Remember that schemas are used for optimization and for checking update validity. Simplification of DTDs can be done for query optimization without compromising correctness, but the same is not true for update validity.

¹¹The document in Example 1 specifies a tree. But XML specification allows for two special kinds of attribute types - ID and IDREF. They provide (definition, reference) chaining of XML element structures different from the hierarchical element structure. Only when these attributes are used, the XML document represents a graph, and not a tree.

¹²id is a better term than address, but we decided to use address so as not to conflict with the XML id declaration.

3 Overview of *QuiXote*'s Data and Storage Model

In this section, we describe several aspects of the *QuiXote* system: the QNX data model, the properties defined for an element, and the storage model used within *QuiXote*. The QNX data model views an XML repository as a set of $\langle \text{schema}, \text{setOfData} \rangle$ pairs. In other words, for each schema, we maintain the set of documents which conform to the schema. The data in an XML document is viewed as a graph, and “edges” between graphs are used to represent inter-document links. Queries are based on this QNX data model. The storage model describes how the document is stored. *QuiXote* stores the XML document as a tree. Link attributes (such as IDs and IDREFs) are used to capture the graph view of the document supported by the data model.

3.1 The QNX Data Model

The QNX data model views an XML repository as a set of $\langle \text{schema}, \text{setOfData} \rangle$ pairs, where for every schema, we maintain the set of documents that conform to the schema.¹³ We can therefore represent an XML repository as

$$R = \{ \langle s_1, D_1 \rangle, \langle s_2, D_2 \rangle, \dots, \langle s_n, D_n \rangle \}$$

Here R represents the repository, $s_i, 1 \leq i \leq n$ is the set of schemas in R , and D_i is the set of documents in R which conform to s_i (that is, the schema for any document $d \in D_i = s_i$).

In the QNX data model, a schema in the repository is viewed as a graph rooted at the DOCTYPE element as shown in Figure 1. This captures the entire DTD semantics. QNX views an XML document also as a directed graph, rooted at the document root element as shown in Figure 2. This graphical view is obtained from the XML document as follows.

```
<library>
  <setOfAuthors>
    <author id="a1" firstName="Charles" middleInitial="F" lastName="Goldfarb"/>
    <author id="a2" firstName="Paul" lastName="Prescod"/>
  </setOfAuthors>
  <setOfBooks>
    <book author="a1 a2">
      <title>
        The
        <em>XML</em>
        Handbook
      </title>
    </book>
  </setOfBooks>
</library>
```

Example 2 A document which specifies intra-document referencing.

An XML document D can be thought of as defining three sets of nodes - a set of element nodes denoted by E , a set of attribute nodes denoted by A and a set of text nodes denoted by T . One node of this set of element nodes, $r \in E$ is defined as the document root element. XML defines two sets of properties for each element $e \in E$, which we denote by (P, R) . P defines an ordered list of parent-child relationships in D where the parent is e (the child can be another element node or a text node). R defines the set of attributes for e . Attribute nodes are used for defining attribute values and also for specifying intra-document links. Therefore we define an attribute node by (attribute-name,

¹³This schema can be either defined by the document, or extracted from the document.

attribute-value, links), where links refer to an ordered list of elements defined by the attribute-value. Table 1 shows five element nodes for the document in Example 2 and their properties.

Element address	Element tag name	Properties
1	library	$P = \langle 1.1, 1.2 \rangle$ $R = \{\}$
1.1.2	author	$P = \langle \rangle$ $R = \{(id, a1, \langle \rangle), (firstName, Paul, \langle \rangle), (lastName, Prescod, \langle \rangle)\}$
1.2	setOfBooks	$P = \langle 1.2.1 \rangle$ $R = \{\}$
1.2.1	book	$P = \langle \rangle$ $R = \{(author, a1 a2, \langle 1.1.1, 1.1.2 \rangle)\}$
1.2.1.1	title	$P = \langle \text{"The"}, 1.2.1.1.1, \text{"Handbook"} \rangle$ $R = \{\}$

Table 1 The different parent-child as well as attribute properties for 5 elements in the document in Example 2. The parent-child relationships form an ordered list, whereas the attribute relationships form a set.

Given such a document D , we obtain the graphical view $G = (V, E)$ as follows. V is the set of element, attribute as well as text nodes in D . We denote the set of element nodes in V by E , the set of attribute nodes by A , and the set of text nodes by T .¹⁴ The edges in E are directed edges and are defined as follows. For an element node $e \in E$, we define (a) an ordered list of edges, P to every element node and text node that is a child of e . (b) a set of edges R to every attribute node of e .

For an attribute node a , we define an edge, only if it is a link attribute for example, attributes of type IDREF. For such link attributes, we define an ordered list of directed edges, L from a to all element nodes referenced. No edges are defined from a text node. Figure 3 illustrates the graph view defined by the QNX data model for a portion of the XML document in Example 2.

The QNX data model views a document as a graph, and thus it captures intra-document referencing. But an XML repository can consist of inter-document referencing as well (XLink). By defining “edges” between graphs, the QNX data model captures cross-document referencing.

3.2 Properties defined for an Element

Any query processing system has to define queryable elements and their properties. These queryable elements are tuples for a relational database and objects for an object data base. *QuiXote* defines elements in the document as basic elements for querying. For an element A , we define the following properties:

- *Element properties* $\mathcal{E}(A)$ All elements in the sub tree, say ST , rooted at A . Every such element has a depth associated with it. The element-properties of an element form an ordered list of (element, depth) pairs, where the order is defined as the order in which the elements are seen during the pre-order traversal of the sub tree, ST .
- *Attribute properties* $\mathcal{A}(A)$ The attributes (in XML jargon) of A .¹⁵ *QuiXote* views the attribute-properties of an element as a set of (attribute-name, attribute-value, links) items. The “links” is defined for link attributes, and represents an ordered list of the elements referenced by this attribute.

¹⁴ (E, A, T) forms a partition of V , because there is no overlap between two sets of nodes. Also V , A and T correspond to the nodes declared by XML.

¹⁵The attributes of an element are identified uniquely by the attribute names.

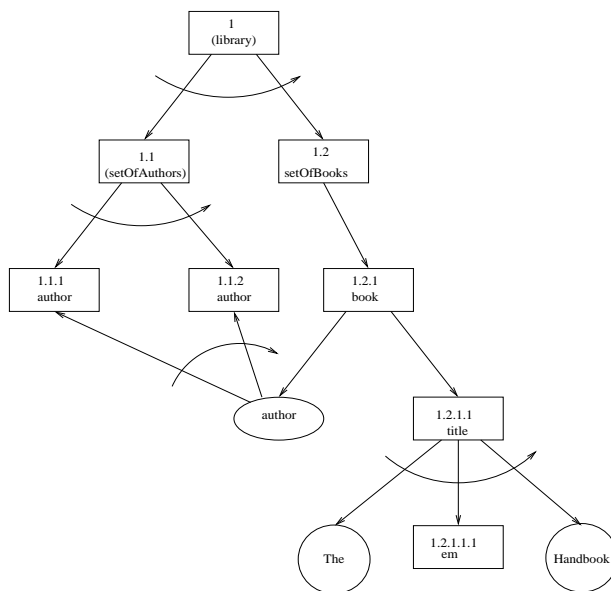


Figure 3: The graphical view of the document in Example 2 in the QNX data model. Each element node has an ordered list of edges representing its child element as well as text nodes. The attribute author of book is of type IDREFs and has an ordered set of edges to two elements. The arc direction represents the order among the edges.

- *Text properties* $\mathcal{T}(A)$ The text contained in the text nodes in the subtree rooted at A . The text-properties of an element are viewed as an ordered list of (text-value, depth) pairs, where the order is defined as the order in which the text nodes are seen during the pre-order traversal of ST .

The element properties allow queries on the structure contained in the document and the attribute and text properties allow queries on the text contained in the document. The definition of element properties and text properties in this model allows two different elements to share element and text properties. This is not true for attribute properties. An element property (or a text property) can be shared between two elements A and B , only if A is an ancestor of B or vice versa. Therefore one can uniquely identify element or text properties by using a tuple $\langle \text{element (or text node), depth} \rangle$. Table 2 illustrates how *QuiXote* defines properties for a few elements for the document in Example 2.

Element address	Properties
1	$\mathcal{E}(1) = \langle (1.1, 1), (1.1.1, 2), (1.1.2, 2), (1.2, 1), (1.2.1, 2), (1.2.1.1, 3), (1.2.1.1.1, 4) \rangle$ $\mathcal{A}(1) = \{ \}$ $\mathcal{T}(1) = \langle (\text{The}, 4), (\text{XML}, 5), (\text{Handbook}, 4) \rangle$
1.2.1	$\mathcal{E}(1.2.1) = \langle (1.2.1.1, 1), (1.2.1.1.1, 2) \rangle$ $\mathcal{A}(1.2.1) = \{ (\text{author}, a1 \ a2, \langle 1.1.1, 1.1.2 \rangle) \}$ $\mathcal{T}(1.2.1) = \langle (\text{The}, 2), (\text{XML}, 3), (\text{Handbook}, 2) \rangle$

Table 2 This table illustrates how *QuiXote* defines properties for elements. Note the definition of element and text properties in a depth first order. The attribute properties form a set.

3.3 Storing XML documents in QuiXote

In this subsection, we describe how *QuiXote* stores XML documents to support the QNX data model. Previous work has considered storing XML documents as graphs in an object-oriented database [10, ?], or as flat relations in a relational database [28, ?]. The storage model used determines the efficiency for querying parent-child and link relationships. When XML documents are stored as graphs, parent-child as well as link relationships are captured explicitly. Therefore, queries on these relationships are supported efficiently. But there is significant overhead in processing the document as a graph with possible cycles. For example, the complexity of constructing structural summaries (dataguides in [12]) can be exponential for a graph (it is linear for a tree). On the other hand, if we store an XML document as a relational database, the efficiency of querying on relationships decreases tremendously, since (expensive) joins have to be performed to process these queries.

QuiXote stores an XML document as a tree. Here, parent-child relationships are captured explicitly, but link relationships are captured implicitly using link attributes. Thus, querying on parent-child relationships is very efficient, but querying on link relationships will require joins. *QuiXote* relies on various indexing schemes to improve the efficiency of these queries.¹⁶

QuiXote uses a binary storage model called *Millau* [11] to store the XML tree. *Millau* achieves good compression, while still maintaining the structure of the document. This is done using tokens for element and attribute names specified in the DTD and using standard text compression for attribute and text values. Experiments show that *Millau* can achieve compressions of about 80%.

Most XML parsers load an XML document entirely into virtual memory before it allows navigation of the document. This imposes restrictions on the size of the document, and implies a huge overhead for loading a large document. To overcome this restriction, *Millau* defines a load model, where portions of the document is loaded into memory as and when needed.

Other systems exist for compressing, storing and loading XML documents. XMill [15] is a compression scheme for XML. Given an XML document, XMill separates structure from content, groups the content into different containers based on meaning, and compresses each container separately. XMill achieves greater compression than *Millau*, but at a significant overhead for compression and decompression. PDOM [22] provides a binary load and store model for XML, where documents are stored as trees. It also provides the capability to load only portions of the document. But PDOM achieves much less compression as compared to XMill or *Millau*.

4 The QNX Query Language

In this section, we will first discuss requirements that we consider important for an XML query language. Following, we will introduce the QNX query language, and discuss its features.

4.1 Requirements for an XML query language

Recently, different declarative query languages have been proposed for XML. We classify the different query languages into two categories:

- Query languages based on *path expressions*, such as Lorel [2] and XQL [24, ?].
- Query languages based on *tree structures*, such as XML-QL [5].

¹⁶XML [31] and DOM [7] specifications support the tree model. Therefore, when we store an XML document as a tree, we are more likely to obey the semantics defined by these standards specifications. For example, we still maintain a single parent node for any node (but when we would store it as a graph, we might allow multiple parents for a node). Also it is easier to maintain update semantics as given by the specifications if we store the document as a tree.

Two factors mostly considered in designing a query language are its expressive power and its ease of use. Since a tree structure can always be decomposed into a set of paths, a query language that supports *multiple path expressions* has the same expressive power as one that supports tree structures. Lorel and XQL support multiple path expressions; therefore, all the three query languages have the same expressive power. When we considered ease of use, we observed that queries that do not specify conditions on siblings typically tend to be shorter if path expressions are used. But tree structures are easier to specify conditions on siblings.¹⁷

From our experience in querying XML, designing a pattern match based query language for XML (PatML [27]) and using *QuiXote* for mining XML documents, we list the following features that we consider essential in an XML query language.¹⁸

- An XML query language should support at least two types of returning elements - *deep return*, which returns the subtree rooted at the element, and *shallow return*, which returns an element (with or without attributes). XQL supports this feature. To the best of our knowledge, none of the other query languages provide this capability.
- The restructuring of documents should also be supported by an XML query language, since many queries require the query result document to have a different structure than the source document.
- Along with restructuring, the query language should allow specification of grouping result elements. This is typically supported using variable binding and Skolem functions [1, ?] (both Lore and XML-QL support this feature).
- The query language should also support specifications of regular path expressions for reaching an element. This is because there can be cycles in the schema.¹⁹
- Since an XML document represents order among child nodes, a query language should support queries on the order among child nodes.
- The content of an XML document (text values and attribute values) is not typed. All attribute values are treated as strings. Therefore, to perform meaningful operations, a query language should try to identify the data types associated with the different content values.
- The query language should also support common database constructs for supporting set operations (union, difference, etc.) and joins.
- It should support constructs for specifying the semantic view as well as the literal view for links (both intra-document and inter-document).
- The query language should support at least three constructs for querying text content (text nodes) - (a) query the value of each text node individually, (b) combine the text values of all text nodes at a particular depth from an element, use the order specified in the document and query this combined value, (c) combine the text values of all text nodes in the subtree that is rooted at an element in a dfs manner, and query these values.²⁰

¹⁷XSLT [33] and XPath [32] have the beginnings of a query mechanism. But they do not provide all the functionality provided by a declarative query language - queries based on set operations (union, intersection), join queries etc.

¹⁸Most of our observations here are based on observations in QL'98 [23].

¹⁹If we consider the DTD in Example 1, a user might want to select all editors of monographs. The query for this in an OQL type syntax will be `(?)*.monograph.editor`, where `?` can match any element.

²⁰Constructs for (a) will be used to perform queries such as select books published by McGraw-Hill in Example 1. Constructs for (b) can be used to look for section titles in an article, when we do not know the parent element of these section title values. Constructs for (c) will be used in a formatted text, where markup (tags) is used for specifying format.

4.2 QNX

We designed the QNX query language based on the observations, principles and requirements mentioned in the last section. An example query in QNX is given below.

Query Q in English

Select book elements from the document library.XML which are published by McGraw-Hill and which have an author element as a child. The above query as expressed in QNX is.

Query Q in QNX

```
<Query qnx:QUERY='Project'>
  <FROM Source='library.XML' />
  <qnx:PATTERN>
    <book>
      <publisher>
        <qnx:PCDATA qnx:OPERATOR='eq' qnx:VALMATCH='McGraw-Hill' />
      </publisher>
      <author />
    </book>
  </qnx:PATTERN>
</Query>
```

Query Q in Lorel

```
SELECT Z
FROM book B in library.XML
WHERE  $\exists$  (B.publisher  $P_1$ ) and  $\exists$  ( $P_1$ .text T and T.value = 'McGraw-Hill') and  $\exists$  (B.author  $A_1$ )
```

Result set for Q

```
<book>
  <publisher>McGraw-Hill</publisher>
  <author firstName='Steven' lastName='Holzner'>
</book>
```

Example 3 Example Query in QNX

In the Lorel query, Z specifies the result. This is because QNX returns a set of elements as the result set for the entire query. The result set is the set of all elements on which conditions are specified in the query. If no output structure is specified QNX uses a default result structure. This structure is the same as that specified in the query.

The main features of QNX are listed below.

- QNX has an XML syntax, i.e. a query in QNX will be a well-formed XML document. This has the advantage, that any existing XML parser can be used for parsing a QNX query.
- QNX is a DTD-based query language. In other words, the DTD for the query language can be obtained as a function of the DTDs of the subject documents (i.e. the documents to be queried). Therefore, meaningless queries can be avoided during query formulation itself.
- QNX is closed under a DTD algebra. In other words, an output DTD can be obtained for every query as a *function of the query and the DTDs of the source documents*.

- A QNX query can specify multiple subject documents that are used as the basis for a query.
- When query conditions are specified on siblings, QNX assumes that the query does not specify any order among the siblings, unless mentioned specifically. This is because of our observation that DTDs are often ordered unnecessarily.²¹

5 QuiXote System Architecture

The *QuiXote* query processing system processes queries in the QNX query language. It uses schema (DTD) extensively for query optimization. The basic architecture of *QuiXote* is shown in Figure 4. *QuiXote* consists of two main components - a *preprocessor* and a *query processor*. The preprocessor first extracts the schema for XML documents which do not have a pre-defined schema. It then computes relationship sets between elements and attributes from the schema. Various index structures are also built from the data contained in the document. The query processor accepts queries in the QNX query language, generates “optimal” query execution plans for each source document specified in the query, executes the query execution plans generated and returns the result set (as an XML document) to the user. This section describes the two components in detail.

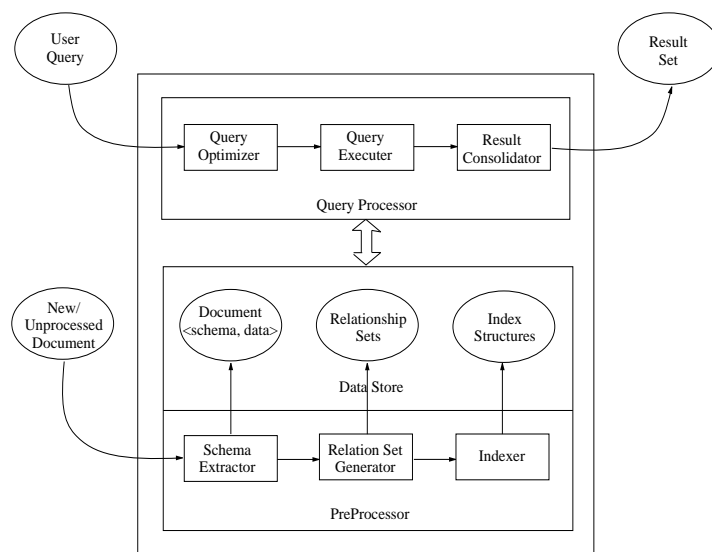


Figure 4: The *QuiXote* System Architecture. The relationship sets and the indices computed by the preprocessor are used by the query processor to execute the user query efficiently.

5.1 QuiXote Preprocessor

The preprocessor is used to optimize query execution time by pre-processing incoming XML documents. The structure of the preprocessor is shown in Figure 5. First the *Schema Extractor* extracts the DTD for every new or unprocessed document which does not have a pre-defined schema. The DTD and the data in the document are then used by the other components of the preprocessor.

- The *Relation Set Generator* computes relationship sets between elements and attributes from the DTD (either defined or discovered).

²¹A query such as select books that have an author defined and a publisher defined, typically does not require that the publisher be a right sibling of author or vice versa.

- The *Indexer* builds structure, text and link indices from the document data.

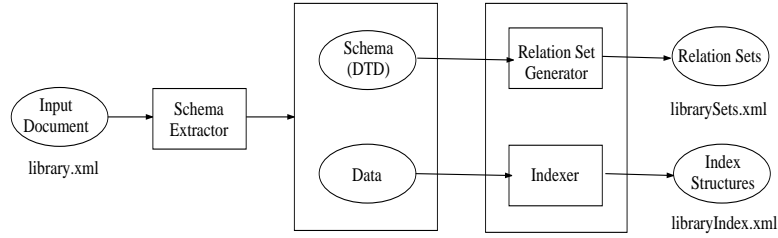


Figure 5: *QuiXote* PreProcessor: The schema extractor extracts the schema, the relation set generator computes relationship sets, the indexer builds index structures from the document data

5.1.1 Relation Set Generator

The relation set generator computes relationship sets from the schema. These relationship sets are later used for query optimization and processing. Computing relationship sets from the DTD has been studied earlier. In [3], three kinds of relationship sets, *obligation*, *exclusivity* and *entrance location* are computed. For an element A , the obligation set gives the descendants that should always be present for A , the exclusivity set gives the set of ancestors that should always be present for A . The entrance location set for a path (A, B) gives the set of elements that should always be present in the path. *QuiXote* extends the notion of these sets by maintaining level information with each set. *QuiXote* also introduces five additional relationship sets, *child*, *parent*, *ancestor*, *attribute*, and *reachability*. The child set is the set of possible child elements and the corresponding parent-child relationship type (this can be one of $\{*, +, ?, ONCE\}$), the parent set is a set of its possible parent elements, the ancestor set gives the set of possible ancestors for A . The attribute set maintains the possible attributes A can have and their characteristics as defined by the DTD. The reachability set gives the set of possible descendants A can have in an XML document conforming to the DTD. A formal description of these relationship sets and how *QuiXote* determines them is given below.

- *Child*

The child relationship set for an element A denoted by $C(A)$ is given by

$$C(A) = \{(C_1, T_1), (C_2, T_2), \dots, (C_n, T_n)\}$$

where C_1, C_2, \dots, C_n represent all the child elements which A can have. T_i associated with a C_i denotes the derived parent-child relationship between A and C_i [28]. This can be either $+, *, ?$ or *ONCE*. *ONCE* is used to denote a child that can occur exactly once.

Two sample child sets for the document in Example 1 are

$$C(\text{library}) = \{(\text{book}, *), (\text{monograph}, *)\}$$

$$C(\text{title}) = \{(\text{PCDATA}, \text{ONCE})\}$$

- *Parent*

The parent relationship set for an element A , denoted by $P(A)$ is defined as

$$P(A) = \{P_1, P_2, \dots, P_n\}$$

where $P_i, 1 \leq i \leq n$ represents a possible parent element for A .

One of the parent relationship sets obtained from the DTD in Example 1 is

$$P(\text{title}) = \{\text{book}, \text{monograph}\}$$

- *Attribute*

The attribute relation set for an element A is given by

$$\text{Attr}(A) = \{(At_1, Dec_1, Def_1, Val_1), (At_2, Dec_2, Def_2, Val_2), \dots, (At_n, Dec_n, Def_n, Val_n)\}$$

where At_1, At_2, \dots, At_n are the name of all the attributes which A can have. Dec_i denotes the declared type of At_i , Def_i denotes the default type of At_i and Val_i denotes the default string value defined for that attribute (if any)²². For the document in Example 1, the attribute set for *author* is given by

$$\begin{aligned} \text{Attr}(\text{author}) = \{ & (\text{firstName}, \text{CDATA}, \text{REQUIRED}, \phi), \\ & (\text{middleInitial}, \text{CDATA}, \text{IMPLIED}, \phi), \\ & (\text{lastName}, \text{REQUIRED}, \phi) \} \end{aligned}$$

- *Reachability*

The reachability set for an element A , denoted by $R(A)$ is given by

$$R(A) = \{(A_1, L_1), (A_2, L_2), \dots, (A_n, L_n)\}$$

where A_1, A_2, \dots, A_n are possible descendants of A as computed from the DTD graph. The L_i associated with A_i represent the set of levels at which A_i can be present as a descendant of A . The reachability set for the leaf nodes in the graph is the empty set. For convenience of explanation, we define a set $R'(A)$ which is obtained from $R(A)$ by omitting the level information.

$$R'(A) = \{A_1, A_2, \dots, A_n\}$$

The set $R'(A)$ represents the set of elements which can be present as a descendant of A in a document conforming to the DTD. For an element A which has n children denoted by $S = \{C_1, C_2, \dots, C_n\}$, the recurrence relation used to obtain the set $R'(A)$ is given by

$$R'(A) = C_1 \cup R'(C_1) \cup C_2 \cup R'(C_2) \cup \dots \cup C_n \cup R'(C_n)$$

The level information for an element $A_i \in R'(A)$ is obtained as follows. Let $D_1, D_2, \dots, D_k \subseteq S$ be such that $A_i \in R'(D_j) \mid 1 \leq j \leq k$. Let $L_i(B)$ represent the set of levels associated with A_i in the reachability set for B . Then

$$L_i(A) = L'_i(D_1) \cup L'_i(D_2) \cup \dots \cup L'_i(D_k)$$

$$L'_i(E) = \{e_1 + 1, e_2 + 1, \dots, e_p + 1\} \mid L_i(E) = \{e_1, e_2, \dots, e_p\}$$

The reachability set for two elements in the DTD shown in Figure 1 are given below. Note the use of *ANY* to take care of cycles in the DTD graph.

- $R(\text{library}) = \{(\text{book}, \{1\}), (\text{monograph}, \{\text{ANY}\}), (\text{title}, \{\text{ANY}\}), (\text{author}, \{2\}), (\text{publisher}, \{\text{ANY}\}), (\text{PCDATA}, \{\text{ANY}\}), (\text{editor}, \{\text{ANY}\})\}$
- $R(\text{title}) = \{(\text{PCDATA}, \{1\})\}$

- *Ancestor*

The ancestor relationship set for an element A denoted by $G(A)$ is given by

$$G(A) = \{(P_1, L_1), (P_2, L_2), \dots, (P_n, L_n)\}$$

where $P_i, 1 \leq i \leq n$ represents a possible ancestor element which can be present in the document. The level information, L_i associated with an ancestor, P_i represents the set of levels at which P_i can be present as an ancestor with respect to A . The ancestor relationship sets can be computed from the reachability relationship sets as follows. If $(E_i, L_i) \in R(A)$, then $(A, L_i) \in G(E_i)$.

²²Refer to [8] for a description of the default type, declared type and the default string value for an attribute.

An ancestor relationship set obtained for Example 1 is

$$G(\textit{title}) = \{(\text{book}, \{1\}), (\text{monograph}, \{\text{ANY}\}), (\text{library}, \{\text{ANY}\}), (\text{editor}, \{\text{ANY}\})\}$$

- *Obligation*

The obligation set for an element A denoted by $O(A)$ is given by

$$O(A) = \{(A_1, L_1), (A_2, L_2), \dots, (A_n, L_n)\}$$

where A_1, A_2, \dots, A_n are descendants of A that must be present in every document conforming to the DTD. The L_i associated with A_i represents the set of levels at which A_i can be present as a descendant of A . This is the same as the set of levels present in the reachability set. As for the reachability set, we define a set $O'(A)$ which is obtained from $O(A)$ by omitting the level information.

$$O'(A) = \{A_1, A_2, \dots, A_n\}$$

Consider an example in which element A 's content model is given by

$$A \rightarrow (((C_1|C_2), C_3)|C_4^?), C_5^*, C_2)$$

In computing the obligation set of an element A , content model subtrees with the $*$ or $?$ operators are ignored. This reduces the above content model to $((C_1|C_2), C_3), C_2)$. Then an AND-OR tree is constructed with *AND* representing “,” and *OR* representing “|” with the elements as the leafs.

The obligation set $O''(N)$ for a node N in the AND-OR tree is computed recursively as follows.

- For an element node (leaf node), $O''(N) = O'(N)$.
- For a node N , we define $Contrib(N) = O''(N) \cup N$ if N is a leaf node, and $O''(N)$, otherwise.
- For an AND node N , with n child nodes C_1, C_2, \dots, C_n (the child nodes can be OR nodes or element nodes), the obligation set is given by $O''(N) = \bigcup_{i=0}^n Contrib(C_i)$.
- For an OR node N , with n child nodes C_1, C_2, \dots, C_n (the child nodes can be AND nodes or element nodes), the obligation set is given by $O''(N) = \bigcap_{i=0}^n Contrib(C_i)$.

Two sample obligation sets from Example 1 are

$$O(\textit{library}) = \{\}$$

$$O(\textit{title}) = \{(\text{PCDATA}, \{1\})\}$$

- *Exclusivity*

The exclusivity set for an element A , denoted by $E(A)$ is given by

$$E(A) = \{(A_1, L_1), (A_2, L_2), \dots, (A_n, L_n)\}$$

where A_1, A_2, \dots, A_n are ancestors of A that must be present in every document conforming to the DTD. The level set L_i associated with an element A_i represents the set of levels at which A_i can be present as an ancestor of A . (This is the same as the set of levels at which A can be present as a descendant of A_i obtained in the reachability sets.) Let $E'(A)$ denote the exclusivity relationship set for A without the level information, that is $E'(A) = \{A_1, A_2, \dots, A_n\}$. For an element A with n possible parents, P_1, P_2, \dots, P_n , $E'(A)$ is defined recursively as follows.

$$E'(A) = (P_1 \cup E'(P_1)) \cap (P_2 \cup E'(P_2)) \cap \dots \cap (P_n \cup E'(P_n))$$

Two example exclusivity sets for the DTD in example 1 are

$$E(\textit{title}) = \{(\text{library}, \{\text{ANY}\})\}$$

$$E(\textit{author}) = \{(\text{book}, \{1\}), (\text{library}, \{2\})\}$$

- *Entrance Location*

The entrance location set for an element A , denoted by $EL(A)$ is given by

$$EL(A) = \{(B_1, \{(C_{11}, L_{111}, L_{112}), (C_{12}, L_{121}, L_{122}) \dots (C_{1n_1}, L_{1n_11}, L_{1n_12})\}), \\ (B_2, \{(C_{21}, L_{211}, L_{212}), (C_{22}, L_{221}, L_{222}) \dots (C_{2n_2}, L_{2n_21}, L_{2n_22})\}), \dots, \\ (B_k, \{(C_{k1}, L_{k11}, L_{k12}), (C_{k2}, L_{k21}, L_{k22}) \dots (C_{kn_k}, L_{kn_k1}, L_{kn_k2})\})\}$$

The above equation says that for every occurrence of the path (B_i, A) in the XML document, there always exist the elements $\{C_{i1}, C_{i2}, \dots, C_{in_i}\}$ in the (B_i, A) path. The level information associated with an element C_{ij} is given by two sets of levels. L_{ij1} represent the set of levels at which C_{ij} can be present as a descendant of B_i , and L_{ij2} represent the set of levels at which A can be present as a descendant of C_{ij} . Again the level information is obtained from the reachability sets. Therefore we define $EL'(A)$ as

$$EL'(A) = \{(B_1, \{C_{11}, C_{12} \dots C_{1n_1}\}), \\ (B_2, \{C_{21}, C_{22}, \dots C_{2n_2}\}), \dots, \\ (B_k, \{C_{k1}, C_{k2}, \dots C_{kn_k}\})\}$$

For an element, A with n possible parents, $\{P_1, P_2, \dots, P_n\}$, the entrance location is obtained as follows.

- For every parent P_i , we add $(P_i, \{\})$ to the entrance location relationship set.
- If two parents, P_i and P_j have the terms (B_1, S_1) and (B_1, S_2) respectively in the entrance location sets, then the set for A will contain $(B_1, ((P_i \cup S_1) \cap (P_j \cup S_2)))$.

For the DTD in example 1,

$$EL(author) = \{(library, \{(book, \{1\}, \{1\})\}), \dots\} \\ EL(library) = \{(title, \{\}), \dots\}$$

5.1.2 Indexer

The *QuiXote* indexer maintains three kinds of indices for a document, *value index* [17] corresponding to text, *structure index* corresponding to tree structure patterns, and *link index* corresponding to link relationships. The granularity of indexing is an XML document. In other words separate index structures are maintained for each document in the repository.²³

An indexing scheme is measured in terms of three factors - space overhead, maintenance overhead and access overhead [14]. Space overhead is the additional storage required for storing the index structures. *QuiXote* stores the index structures as a trie [4] and uses a compressed format provided by *Millau*, these reduce the storage cost. Maintenance cost is the cost associated with maintaining an index when updates are made on the data for which the index is built. *QuiXote* does not currently support updates and therefore maintenance costs are not considered.²⁴ Access cost is the overhead in using the index structures for a query. This depends on how the index structures are organized. *QuiXote* maintains the indices as a trie [4] ensuring that the index access cost is linear with respect to the size of the query and independent of the number of indices present. Storing the index structures using *Millau* also helps us to navigate through the index structures in an efficient manner. In this

²³We decided to maintain separate index structures for each document for two main reasons 1) Most queries query on a subset of documents in the repository (for example, select students with Java skills from Stanford resume documents). Detecting this subset of documents can be as easy as checking the name of the documents. Therefore maintaining separate index structures for each document decreases the number of unnecessary elements considered. 2) When a document is updated, index maintenance is easier.

²⁴As *QuiXote* maintains indices corresponding to each document individually, we expect that the maintenance costs on updates will be small.

section, we describe how value and structure indices are maintained by *QuiXote*. Link indices are not considered as they are presently maintained as simple mappings from ID names to elements.

Value Index

A value index can be viewed as a set of (V, S) pairs, where V is a value key (used for retrieval), and S is a set of element references. An element is referenced using its “address” mentioned earlier. Two kinds of value indices are maintained by *QuiXote* in order to support the two kinds of values for XML - text values and attribute values.

- *Text Index*

Text indices are maintained for the values contained in the text nodes of elements. These indices are maintained as a set of $(E, \{(V_1, S_1), (V_2, S_2), \dots, (V_n, S_n)\})$ 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 with tag name E that contain V_i as one of its text values. The search key for retrieval from a text index is the (E, V) pair where E is the element tag name, and V is a text string.

QuiXote maintains three sets of text indices corresponding to the three constructs for querying text content. The three constructs correspond to how the text content of an element A is considered. The text content of A can be 1) set of all text values of text nodes that are children of A , 2) set of all text values, where each value is a concatenation of the text values of text node descendants of A at a particular depth, or 3) a single text value which is obtained by the concatenation of text values of all text node descendants of A in a depth first manner. Figure 6.a shows the first set of text indices for `title` and `publisher` tags in Example 1.

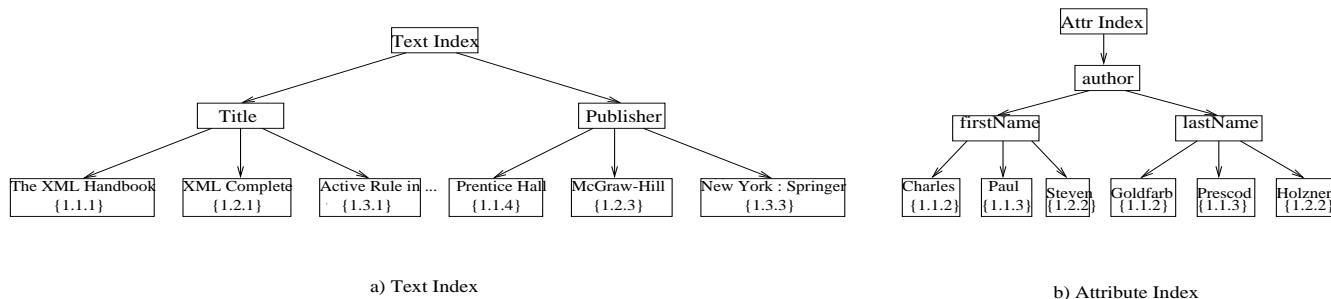


Figure 6: Illustrating Value Indices. a) shows text indices for `title` and `publisher`. For the key (`title`, “XML Complete”), the set of element references is $\{1.2.1\}$. b) shows attribute indices for the attributes `firstName` and `lastName` of `author`. For the key (`author`, `firstName`, “Paul”), the set of element references is $\{1.1.3\}$.

- *Attribute Index*

Attribute indices correspond to attribute values of elements. These indices are maintained as a set of

$$(E, \{(N_1, \{(V_{11}, S_{11}), (V_{12}, S_{12}), \dots, (V_{1n_1}, S_{1n_1})\}), (N_2, \{(V_{21}, S_{21}), (V_{22}, S_{22}), \dots, (V_{2n_2}, S_{2n_2})\}), \dots, (N_k, \{(V_{k1}, S_{k1}), (V_{k2}, S_{k2}), \dots, (V_{kn_k}, S_{kn_k})\})\})$$

pairs, where E is an element tag name, N_1, N_2, \dots, N_k are candidate attributes of E which are indexed, $V_{i1}, V_{i2}, \dots, V_{in_i}$ are possible values that the attribute N_i can take, and S_{ij} is the set of references to elements with tag name E and which have the value V_{ij} for the attribute N_i . The key for retrieval from an attribute index is a triple (E, N, V) , where E is the element tag name, N is the attribute name and V is a text string. An example is shown in Figure 6.b.

Structure Index

Structure indices are maintained corresponding to tree structure patterns and they enable structural queries. These indices are maintained as a set of (Q, S) pairs, where Q is the Euler string [26, ?] representing the pattern tree and S is the set of element references corresponding to this pattern. For the query, *select book elements with at least two authors* the pattern tree is given by

```
<book>
  <author/>
  <author/>
</book>
```

The corresponding Euler String is²⁵

`<book (1), author (2), book (1), author (3), book (1)>`.

QuiXote maintains two kinds of structure indices - *nested index* and *path index* [4]. For a structure query given by the Euler String $S = \langle E_1, E_2, \dots, E_n \rangle$,²⁶ these two indices are described below.

- The *Nested Index* for S is given by the set of all elements in the document (with tag E_1) which match the pattern specified in S .
- The *Path Index* corresponding to position $i, 1 \leq i \leq n$ in S is the set of all elements in the document (with tag E_i) which satisfy the subpattern [27] specified by $S' = \langle E_1, E_2, \dots, E_i \rangle$.

For the document in example 1, let a nested index be created for the query - *find book elements with an author and a publisher*, and a path index be created for the query - *find book elements with at least two authors*. For the path index, let the set of element references be maintained for all positions in the Euler string representing the query. Figure 7 shows the indices created.

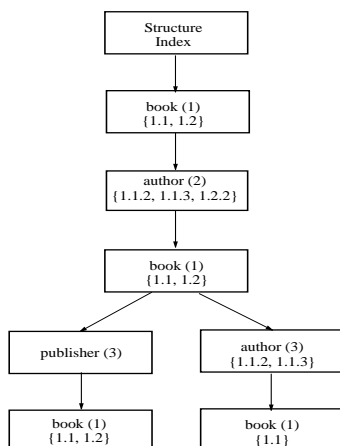


Figure 7: Structure indices maintained as a trie by *QuiXote*. For the structure key given by the Euler string `<book (1), author (2), book (1)>`, the set of element references obtained is $\{1.1, 1.2\}$.

²⁵The numbers used in the Euler string identify multiple occurrence of the same element uniquely.

²⁶Note that $E_1 = E_n$.

5.2 *QuiXote* Query Processor

The *QuiXote* query processor executes QNX queries and returns the result set to the user. It uses the relationship sets and the indices to process queries efficiently. In this section we describe the various components of the query processor. Given a query, the *Source Extractor* first extracts the source documents on which the query will be executed (this is specified in the query). Then the following four operations are performed.

- The *Document Filter* uses the pre-computed relationship sets to eliminate documents that do not qualify for the query. This way the system avoids querying documents that will not contribute to the result set.
- The *Query Optimizer* builds an “optimal” QEP for the document. First the relationship sets are used to perform “strength reduction” (replace complex query constructs by simpler ones and eliminate redundant conditions).²⁷ Candidate query execution plans (which may use indices) are then enumerated for this “reduced” query. The costs for the different QEPs are estimated and an “optimal” QEP is selected.
- The *Query Executer* executes the QEP obtained in the previous step. The QEP specifies a tree with the query operators defined by *QuiXote* as its nodes. This operator tree is executed by the executer and the result set obtained.
- The *Schema Generator* computes the schema (DTD) for the output document (result set). This output schema is computed as a function of the query submitted by the user and the DTD for the source document.²⁸

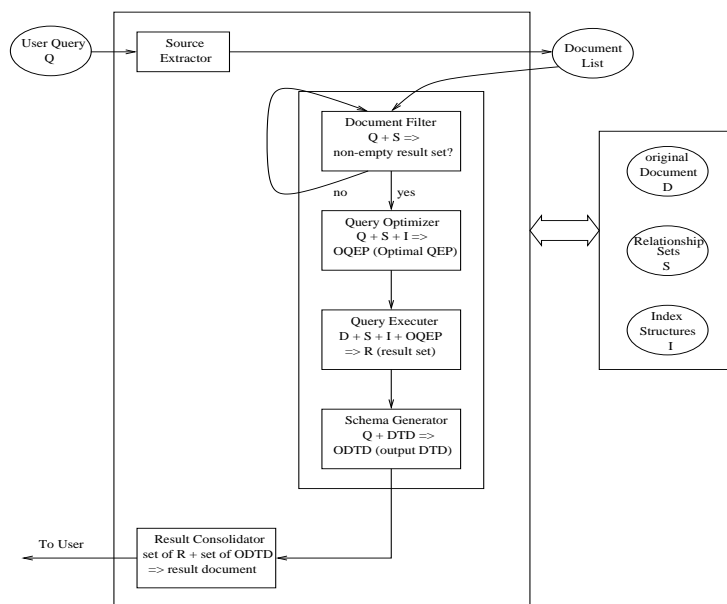


Figure 8: The *QuiXote* Query Processor Architecture. Each document extracted by the source extractor is processed by the document filter, optimizer, executer and schema generator. The result sets from all documents are merged by the result consolidator and sent to the user.

²⁷Lore calls it compile time path expansion.

²⁸The implementation allows the user to specify whether he desires to obtain the output DTD or not.

After the query has been executed on all the specified source documents, the *Result Consolidator* combines all the result sets obtained. It also merges the output DTDs to obtain a DTD which is compatible with all the result sets. The query processor architecture is shown in Figure 8.

5.2.1 Source Extractor

The source extractor obtains the list of source documents on which the query should be executed. A source specified in the query can be either a document in the repository or the result set of a nested query. In the query in example 3, the user identifies `library.xml` as the source and is picked up by the source extractor in a source document list.

5.2.2 Document Filter

The document filter filters out the documents (from the document list extracted by the source extractor) that will produce an empty result set. This uses the relationship sets for the document. Table 3 lists some sample conditions in the query and the corresponding conditions in the relationship set that cause a document to be eliminated. A document that is filtered out is no longer considered for the query. For the above example query, it is possible to get non-empty result set from `library.xml` and therefore this document is sent to the next stage.²⁹

No.	Condition specified in Q	Condition present in $S(D)$ for D to be filtered out
1	$(B, L) \in Des(A)$	$((B, -) \notin R(A) \text{ OR } (((B, L') \in R(A)) \text{ AND } (L \cap L' = \phi)))$
2	$(B, L) \in Anc(A)$	$((B, -) \notin G(A) \text{ OR } (((B, L') \in G(A)) \text{ AND } (L \cap L' = \phi)))$
3	$(B, L) \notin Des(A)$	$((B, L') \in O(A) \text{ AND } (L' \subseteq L))$
4	$(B, L) \notin Anc(A)$	$((B, L') \in E(A) \text{ AND } (L' \subseteq L))$
5	$B, B \in Child(A)$	$((B, ONCE) \in C(A) \text{ OR } ((B, ?) \in C(A)))$
6	$(At, V) \in Attr(A)$	$(At, -, -, -) \notin Attr(A)$

Table 3 Conditions for a source Document D in the document list for query Q to be filtered out. $S(D)$ denotes the relationship sets computed for D . In 1, the query requires that B be a descendant of A at one of the depths in L , but from the reachability sets, we know that B is not a descendant of A at any of the depths in L . The query in 5 requires that A have two child elements with tag name B , but from the child relationship sets, we know that A can have at most one child which is B . The query in 6 requires that A have an attribute At with value V , but from the attribute relationship sets, we know that A cannot have an attribute At .

5.2.3 Query Optimizer

The query optimizer performs two tasks - strength reduction of expensive query constructs, and choosing an optimal query execution plan. Strength reduction involves eliminating redundant query constructs (always true conditions) and replacing expensive constructs (such as all descendants of books that are authors) by simpler ones (such as children of books that are authors).³⁰ A few conditions for using the relationship sets for strength reduction are given in Table 4. For the query in Example 3, a strength reduction can be performed for the root element of the query pattern (book). The query asks to select all books from the document (the book elements can be present at

²⁹It is worth noting that the child, parent, reachability, ancestor and attribute relationship sets are used mostly during this stage. The obligation, exclusivity and entrance location rules can be used only for queries with negation.

³⁰The obligation, exclusivity, entrance location and child sets are typically used for eliminating redundant conditions. The other sets are used only for queries with negation. For simplifying expensive constructs, we use the reachability and ancestor sets.

any depth from the document root element - library) which satisfy the remaining conditions. But from the reachability sets, we know that book elements can be present only at a height of 1 from the document root element, so we need to check only the child elements of library for books.

No.	Condition specified in C	Condition present in $S(D)$	Condition in C'
1	$(B, L) \in Des(A)$	$((B, L') \in O(A)) \text{ AND } (L' \subseteq L)$	ϕ
2	$(B, L) \in Anc(A)$	$((B, L') \in E(A)) \text{ AND } (L' \subseteq L)$	ϕ
3	$(B, L) \in Des(A)$ $\text{AND } (C, L_1) \in Des(B)$	$((C, \{(B, L', L'_1)\}) \in EL(A)) \text{ AND } (L' \subseteq L)$ $\text{AND } (L'_1 \subseteq L_1)$	ϕ
4	$B \in Child(A)$	$((B, +) \in C(A)) \text{ OR } ((B, ONCE) \in C(A))$	ϕ
5	$(B, -) \notin Des(A)$	$(B, -) \notin R(A)$	ϕ
6	$(B, ANY) \in Des(A)$	$(B, L) \in R(A)$	$(B, L) \in Des(A)$
7	$(B, L) \in Anc(A)$	$((B, L') \in E(A)) \text{ AND } (L' \subseteq L)$	ϕ

Table 4 Strength Reduction of the User Query By the Optimizer. Here C represents the condition specified in the original query, $S(D)$ represents the relationship sets for the source document being considered and C' represents the reduced condition corresponding to C . 1, 2, 3, and 4 remove redundant conditions. 5 also removes redundant conditions, but from a query with negation. 6 and 7 show how complex constructs (typically which require traversal to an arbitrary depth or height) are simplified.

Candidate QEPs are then enumerated for the “reduced” query. A cost model is used to estimate the cost of these QEPs. The cost model uses the index access cost, the I/O cost for accessing elements and an estimate of the number of elements to be accessed to estimate the cost of a QEP. An “optimal” QEP is selected and passed on to the query executor for execution.

5.2.4 Query Executor

The query executor executes the QEP obtained from the optimizer and obtains the result set for the document being considered. The QEP specifies a tree of operators with a root node. The query execution starts from this root node. First all the elements in the document which satisfy the conditions for this root node are selected. Then for each selected element, all the conditions specified in the QEP are checked.³¹ The operators which are used by QuiXote in the QEP are described below.

- *Root Operator* returns the document root element. It is represented by $Root(D)$, where D is the document under consideration.
- *Tree Traversal Operators* are used to traverse the source document tree and perform pattern match. For a tree traversal operator V , the subtree rooted at V (say T) specifies a pattern match condition. The candidate elements for V are checked for the pattern match. V returns true if and only if the candidate elements for V matched the pattern specified in T . *QuiXote* uses five different tree traversal operators, which are described below. Let A be the candidate element being considered for pattern match at the parent node of V . (Note that a tree traversal operator is never present as the root node of a QEP and therefore A is always defined.)
 - *Descendant Operators* identify descendant elements of A that satisfy the pattern match. These operators are denoted by $Des(E, N, L, P)$, where E is a set of element names (the

³¹This is a depth first traversal of the QEP tree. Systems such as Lore use breadth first traversal of the QEP. One definite advantage of depth first traversal is the less storage that need to be associated for maintaining bindings.

tag name for a candidate element for V should be specified in E) and L is the set of depths at which a candidate element for V can be present as a descendant of A . N is a count used to indicate how many of the candidate elements for this operator should satisfy the pattern match for V to return true. P specifies the path for reaching a candidate element for V from A . The specification of P is optional, and when no regular path expression is specified, P takes the value ϕ .

- *Ancestor Operators* identify the ancestor elements of A that qualify. An ancestor operator is denoted by $Anc(E, N, L, P)$, where E and N are as explained before. L is the set of heights at which a candidate element can be present as an ancestor of A . The optional P specifies the regular path expression for reaching A from a candidate element for V .
- *Sibling Operators* identify the siblings of A that qualify. They are denoted by $Sib(E, L)$, where E is a set of element names and L is the set of levels at which a candidate element can be present as a sibling of A . Right sibling levels are represented using positive numbers, and left sibling levels using negative numbers.
- *Text Operators* perform match on the value contained in a text node of A . Three different text operators are defined by *QuiXote*.
 - * $Text1(Op, S)$ denotes the text operator used to check the value contained in any individual text node of A . Op is an operator, and S is a string or variable reference. This operator returns true if and only if A has a text node with value Val , such that $Val(Op)S$ is true.³²
 - * $Text2(l, Op, S)$ denotes the text operator used to check the text value obtained by concatenating all the text node descendants of A at a height specified by l .
 - * $Text3(Op, S)$ is used to check the text value obtained by concatenating all the text node descendants of A in a depth first order.
- *Attribute Operators* perform match on the attribute values of A . These operators are denoted by $Attr(At, Op, S)$. This operator returns true if and only if A has an attribute named At with value Val which satisfies $Val(Op)S$. (It also uses type coercion.)
- *Logical Operators* are *AND*, *OR* and *NOT* operators. A logical operator combines the boolean pattern match results of its children.
- *Aggregate Operators* supported at present are *Count* and *GroupBy* operators. The *Count* operator is specified as $Count(E_1, E_2, V)$, and it returns the number of children of E_1 that have tag E_2 and sets V to this value. The *Group Operator* is specified as $Group(E_1, S)$, and it groups all the children of E_1 based on the values for the attributes in S , and returns this grouped element.
- *Index Scan Operator* is used to retrieve elements from a specified index structure. It is denoted by $Index(F, S, K)$, where F is the index file name, S is the kind of index (this can be structure, text, or attribute) and K is the key used to obtain the set of element references. This operator looks up the specified index structures for the key K and obtains the set of element references. It then looks up the source document and retrieves the elements corresponding to these references.
- *Join Operator* combines the results obtained from its child operators. This operator is denoted by $Join(C)$, where C is a condition on which the join is performed. (This operator cannot be a leaf in the QEP).

A QEP composed of the above operators can be classified into one of the following three types. This classification is based on the operators used in the QEP. Figure 9 shows three different QEPs belonging to the three types for the query in Example 3.

³²All text operators use type coercion and cast Val and S to the appropriate type before performing the comparison.

- A *Simple* QEP does not contain ancestor and join operators.
- A *Hybrid* contains at least one ancestor operator, but it does not contain join operators.
- A *Multiple* QEP has at least one join operator.

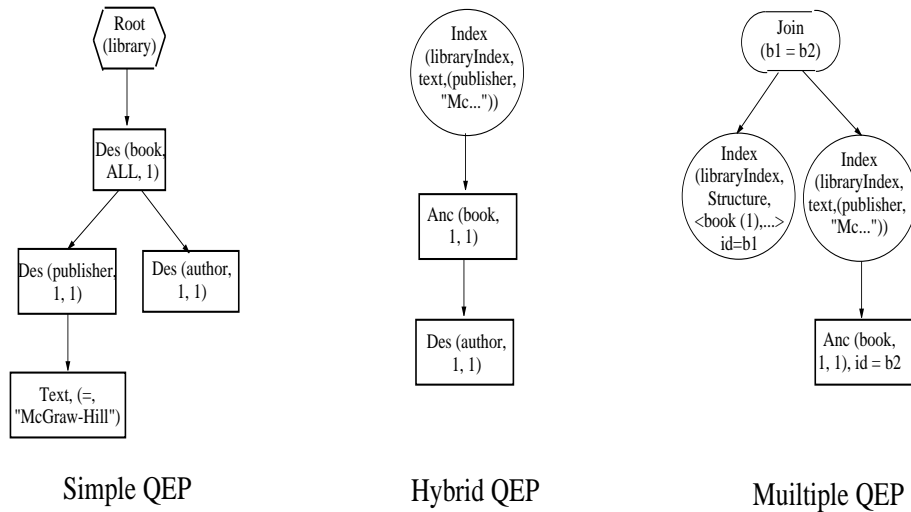


Figure 9: Simple, Hybrid and Multiple QEPs. The Simple QEP does not consist of ancestor or join operators, the Hybrid QEP consists of ancestor operators, but no join, and the multiple QEP has a join operator at the root

A result set is obtained from the execution of the QEP. All elements obtained from the query execution are wrapped under a `qnx:Result` element to form the result set. If output transformation is specified by the user, it is used to obtain the result set. Otherwise a default result set is obtained which consists of all the elements and properties on which the query conditions are evaluated. The structure of this default result set is the same as that specified in the user query. For the query in example 3, no output transformation was specified. The result set when this query is executed on `library.XML` is given by

```
<qnx:Result>
  <book>
    <publisher>McGraw-Hill< /publisher>
    <author firstName='Steven' lastName='Holzner' />
  < /book>
< /qnx:Result>
```

5.2.5 Schema Generator

The schema generator generates the DTD for the result set produced when the query is executed on the document under consideration. This output DTD is obtained as a function of the user query and the DTD for the source document. For the above query, the output DTD for `library.xml` is given by

```
<!DOCTYPE qnx:Result [
  <!ELEMENT qnx:Result (book*)>
  <!ELEMENT book (publisher, author)>
  <!ELEMENT publisher (PCDATA)>
```

```

<!ELEMENT author EMPTY>
<!ATTLIST author firstName CDATA #REQUIRED middleInitial CDATA #IMPLIED
  lastName CDATA #REQUIRED>
]>

```

The above steps are repeated for each document that was extracted by the source extractor.

5.2.6 Result Consolidator

The Result Consolidator combines the result sets obtained for each document by wrapping all the `qnx:Result` elements under a `qnx:setOfResults` element. The output DTDs computed by the Schema Generator are merged and a DTD for the result document is obtained. This result document along with the output DTD is sent back to the user. For the above example query, the result document which the user receives is

```

<!DOCTYPE qnx:setOfResults [
  <!ELEMENT qnx:setOfResults (qnx:Result)>
  <!ELEMENT qnx:Result (book*)>
  <!ELEMENT book (publisher, author)>
  <!ELEMENT publisher (PCDATA)>
  <!ELEMENT author EMPTY>
  <!ATTLIST author firstName CDATA #REQUIRED middleInitial CDATA #IMPLIED
    lastName CDATA #REQUIRED>
]>

<qnx:setOfResults>
  <qnx:Result>
    <book>
      <publisher>McGraw-Hill</publisher>
      <author firstName='Steven' lastName='Holzner'/>
    </book>
  </qnx:Result>
</qnx:setOfResults>

```

6 Performance evaluation

We performed various experiments to evaluate the *QuiXote* system. The first set of experiments were aimed at a basic evaluation, where we ensured the scalability of *Millau*. We also ensured that *QuiXote* can support nested object navigation more efficiently than a relational system. We then evaluated the performance gain achieved from the various optimization strategies which *QuiXote* uses - schema-based optimization (using relationship sets) as well as index-based optimization. We observed that significant performance gains can be achieved using these strategies.

6.1 Basic Evaluation of *QuiXote*

This set of experiments were performed to ensure scalability of the *Millau* storage and load model, and to ensure the advantage of maintaining explicit parent-child relationships. To ensure the scalability of *Millau*, we constructed documents with different sizes (number of elements) and loaded them using *Millau*. We observed that *Millau* took the same time to load a document irrespective of the size of the

document. We then compared the efficiency of the caching and prefetching schemes used by *Millau*. For this, we compared the time required to access all elements in a document for *Millau*, XML4J and MySQL (a relational system). We observed that the access time for *Millau* is comparable to that for MySQL.

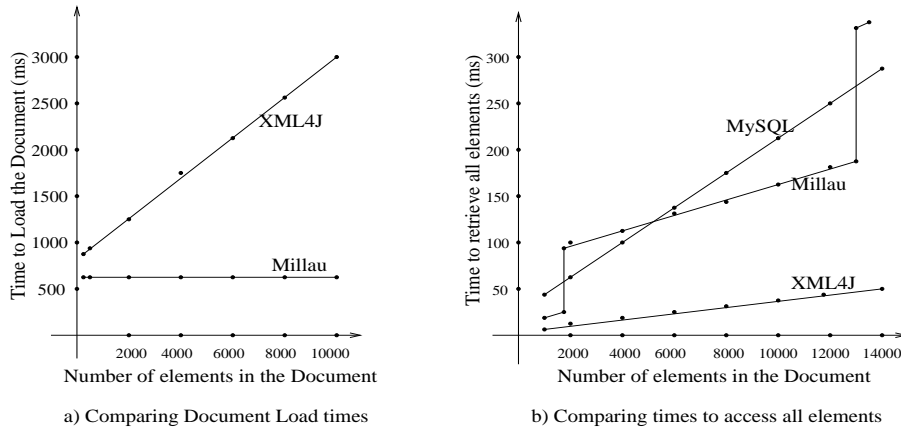


Figure 10: Comparing *Millau*, XML4J and MySQL. a) The document load time for *Millau* is constant irrespective of size of the document. This is because *Millau* loads portions of the document as and when needed. The document load time for XML4J is proportional to the number of elements in the document, as it loads the entire document. b) The time for accessing all elements for XML4J is the least, as the entire document is already loaded into memory. The values for *Millau* is comparable to those for MySQL. The spikes for *Millau* are due to cache misses occurring for these document sizes.

We then wanted to ensure that *QuiXote* can support nested object navigation more efficiently than a relational system. For this we constructed documents with different heights as follows. A document of height 2 consists of a document root element (level 0) and 10,000 child elements (level 1). We added more levels by adding two child nodes to a leaf node of the previous level. Thus a document with level 3 has a document root element (level 0), 10,000 elements at level 1, and 20,000 elements at level 2. The document is mapped into relations as follows - for each level, we constructed a new relation, Each element in that level had an entry in the corresponding relation. Each relation also defined a primary key and an attribute, parentid to capture parent-child relationships. We then observed that the time to access all the leaf elements is less for the relational model. But when we restrict the leaf elements to be selected by specifying a condition on the nodes at level 1, *QuiXote* takes much lesser time. Figure 11 shows the values.

6.2 Performance gain due to optimization strategies

QuiXote uses schema-based optimization (by computing structural relationship sets) as well as index-based optimization. We evaluated the performance gain in using these optimization techniques. The experiments were performed on an IBM NetFinity server, running Windows 2000. The server had a Intel Pentium 3 processor (500MHz), and a 256 MB RAM.

We ran the queries on an XML repository which consisted of 50 XML documents, each document was normalized information extracted from a resume. The documents shared a common DTD. We first computed the relationship sets from this DTD. The effectiveness of relationship sets in filtering

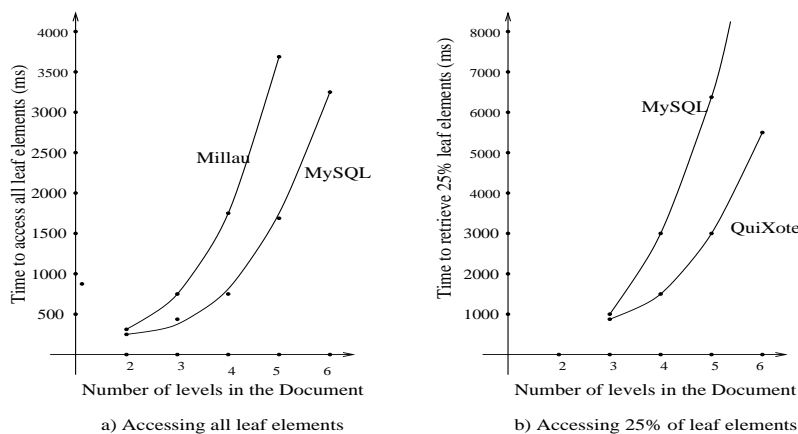


Figure 11: Tree traversal operators versus joins. (a) This shows the time to access all leaf elements when no other conditions are specified. *QuiXote* requires more time, as it has to access all elements (no indices are used). (b) This shows the time to select leaf elements which are descendants of elements at level 1, when a condition is specified for the nodes at level 1. The relational model uses joins to answer this query, but *QuiXote* makes use of the explicit parent-child relationships.

documents and in performing query strength reduction are shown in Table 5 and Table 6 respectively.

No.	Query execution strategy used	Time taken (ms)
1	no relationship sets	2004
2	relationship sets for each document treated separately	1203
3	relationship sets for all documents treated as one	395

Table 5 Performance gain achieved through using relationship sets for filtering documents that will produce an empty result set. The query used had two conditions, one of which was always false. In 1) no relationship sets are used and therefore all documents are loaded. In 2) we load 50 relationship sets, one for each document. In 3) we load only one set of relationship sets for all the 50 documents.

No.	Optimization performed	T_1 (ms)	T_2 (ms)
1	Removing redundant conditions	2537	2453
2	Simplifying complex conditions	2797	2734

Table 6 Performance gain achieved through using relationship sets for strength reduction. T_1 indicates the time taken when no relationship sets were used, and T_2 indicates the time taken when relationship sets were used. The query for 1) had two conditions, one of which was always true. The query for 2) consisted of two conditions, one of which was an element at any depth. But from the relationship sets, we could modify this condition to an element at depth 1 (child).

We then evaluated the performance gain in maintaining value and structure indices. We executed a simple query that selected elements from one document based on one of its attribute values. The time taken to execute this query without any indices was 511ms. With the required attribute index, the query execution took 109 ms. (This included a negligible index lookup cost, 78 ms to retrieve the element from the document and 21 ms to execute the rest of the query and obtain the result.) We then executed a query which specified five child structure conditions and one text condition (on the attribute value of the innermost element). When no structure or value index was present, it took

531 ms to execute the query. When we used the structure index to obtain the element and then check its attribute value, it took 175 ms, and when we used the value index to obtain the element and then traverse the document bottom-up to check the structure conditions, it took 183 ms.

The various values indicate that significant performance gain can be achieved through schema-based optimization and index-based optimization. We could achieve upto 80% performance improvement by using the structural relationship sets for document filter, and also the various index structures. We achieved about 5% performance gain when we used the relationship sets for strength reduction.

7 Conclusions and Future Work

In this paper, we described the architecture of *QuiXote*, a schema driven query processing system for XML repositories. The contributions of *QuiXote* are 1) the QNX data model that can capture the entire DTD semantics, capture the order information as well as the parent-child and link information present in an XML document 2) the QNX query language which is a DTD-based declarative query language, based on a DTD algebra and has an XML syntax 3) using schema-based optimization by computing structural relationship sets and then using them for filtering out documents, and to perform query strength reduction, and 4) using the various index structures to improve query processing efficiency.

XML is an emerging area of importance for web applications, and a number of interesting and challenging research issues in storing, managing, and querying XML are to be solved. Our work in *QuiXote* attempts to address some of these. But significant amount of work has to be done before we achieve a robust, query processing system for XML. The efficiency of *QuiXote* in supporting link relationships is to be determined. For this we would like to evaluate *QuiXote* against repositories whose documents consist of a large number of links. We might need to define more index structures to efficiently support queries on such repositories (for example, we might want to define inverse-link indices for traversing from a ID element to a IDREF element). *QuiXote* also lacks a good cost model for choosing an optimal QEP. We are considering the techniques used by other systems such as Lore (Lore maintains structural summaries for this purpose). *QuiXote* also does not support updates. We would like to have a declarative update language as part of *QuiXote*, here again we could use one of the existing ones.

References

- [1] Abiteboul. S, Kanellakis. P, *Object Identity as a Query Language Primitive*, Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 159-173, 1989.
- [2] Abiteboul. S, Quass. D, McHugh. J, Widom. J, Wiener. J. L, *The Lorel Query Language for Semistructured Data*, International Journal on Digital Libraries, April 1997, 1(1):68-88.
- [3] Bohm. K, Aberer. K, Ozsu. M. T, Gayer. K, *Query Optimization for Structured Documents Based on Knowledge of the Document Type Definition*, Proceedings of IEEE International Forum on Research and Technology Advances in Digital Libraries (ADL'98), Santa Barbara, California, April 1998, pp. 196-205.
- [4] Chen. W, Aberer. K, Neuhold. E. J, *Efficient Algorithms for Determining the Optimal Execution Strategy for Path Queries in OODBS*, Proc. of SEBD'96, San Miniato (Pisa), July 3-5, 1996, pp. 441-452.

- [5] Deutsch. A, Fernandez. M, Florescu. D, Levy. A, Suciu. D, *XML-QL: A Query Language for XML*, *Submission to the World Wide Web Consortium* 19 August, 98, <http://www.w3.org/TR/NOTE-XML-ql>.
- [6] Deutsch. A, Fernandez. M, Suciu. D, *Storing Semistructured Data with STORED*, SIGMOD Conference, Philadelphia, Pennsylvania, June 1-3, 1999, pp. 431-442.
- [7] *Document Object Model (DOM) Level 1 Specification, Version 1.0, W3C Recommendation*, 1 October, 1998 <http://www.w3.org/TR/REC-DOM-Level-1>.
- [8] *W3C XML Specification DTD ("XMLspec")* <http://www.w3.org/XML/1998/06/XMLspec-report-19980910.htm>.
- [9] Enderton. H. B, *A Mathematical Introduction to Logic.*, Academic Press, San Diego, California, 1992.
- [10] Goldman. R, McHugh. J, Widom. J, *From Semistructured Data to XML : Migrating the Lore Data Model and Query Language*, Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99), Philadelphia, Pennsylvania, June 1999.
- [11] Girardot. M, Sundaresan. N, *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, to appear.
- [12] Goldman. R, Widom. J, *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*, Proceedings of the Twenty-Third International Conference on Very Large Data Bases, Athens, Greece, August 1997, pp 436-445.
- [13] Goldman. R, Widom. J, *Approximate DataGuides*, Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Israel, January 1999.
- [14] Hellerstein. J. M, Koutsupias. E, Papadimitriou. C. H, *On the Analysis of Indexing Schemes*, 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, May 1997.
- [15] Liefke. H, Suciu. D, *XMill: an Efficient Compressor for XML Data*, University of Pennsylvania, Department of Computer Science, Technical Reports, 1999.
- [16] Makoto. M, *Forest-regular Languages and Tree-regular Languages*, Unpublished report.
- [17] McHugh. J, Abiteboul. S, Goldman. R, Quass. D, Widom. J, *Lore : A Database Management System for Semistructured Data*, SIGMOD Record, September 1997, 26(3):54-66.
- [18] Milo. T, Suciu. D, *Index Structures for Path Expressions*, 7th International ICDT Conference, Jerusalem, Israel, Jan 10 - 12, 1999, pp. 277-295.
- [19] *The MySQL Homepage* <http://www.mysql.com>.
- [20] McHugh. J, Widom. J, *Compile Time Path Expansion in Lore*, Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Israel, January 1999.
- [21] Nestorov. S, Abiteboul. S, Motwani. R, *Extracting Schema from SemiStructured Data*, SIGMOD Conference 1998, 295-306.

- [22] *GMD-IPSI XQL Engine Version 1.0.2* <http://XML.darmstadt.gmd.de/xql>.
- [23] *QL'98 - The Query Languages Workshop*, <http://www.w3.org/TandS/QL/QL98>.
- [24] Robie. J, *The Design of XQL*, <http://www.texcel.no/whitepapers/xql-design.html>.
- [25] Robie. J, Lapp. J, Schach. D, *XML Query Language (XQL)*, <http://www.w3.org/TandS/QL/QL98/pp/xql.html>.
- [26] Ramesh. B, Ramakrishnan. I. V, *Nonlinear Pattern Matching in Trees*, JI of the Association for Computing Machinery, Vol. 39, No. @, April 1992, pp. 295 - 316.
- [27] Sundaresan. N, Lee. S, Rollins. S, *PatML : A Pattern Match/Replace Language for XML Documents in XML*, IBM Research Technical Report, IBM Almaden Research Center, January 1999.
- [28] Shanmughasundaram. J, Tufte. K, He. G, Zhang. C, DeWitt. D, Naughton. J, *Relational Databases for Querying XML Documents : Limitations and Opportunities*, Proceedings of the 25th VLDB Conference, Edinburg, Scotland, 1999.
- [29] Widom. J, *Data Management for XML Research Directions*, <http://www-db.stanford.edu/widom/xml-whitepaper.html>, Accessed Feb. 14, 1999
- [30] *XML Parser for Java*, <http://www.alphaworks.ibm.com/tech/XML4j>.
- [31] *Extensible Markup Language (XML) 1.0, W3C Recommendation 10-February-1998* <http://www.w3.org/TR/REC-XML>.
- [32] *XML Path Language (XPath) Version 1.0*, W3C Recommendation, 16 November, 1999, <http://www.w3.org/TR/1999/REC-xpath-19991116>
- [33] *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, 16 November, 1999, <http://www.w3.org/TR/1999/REC-xslt-19991116>