XIQS: An XML Indexing and Query System

Shubhashree Venkatesh and Gongzhu Hu (hu@cps.cmich.edu)
Department of Computer Science, Central Michigan University
Mount Pleasant, MI 48859, U.S.A.

Abstract

Retrieval from XML data sets is an actively researched field that presents some different problems from retrieval of relational databases. The challenges stem from the characteristics of the tree structures of XML data. In this paper we present a system, XIQS, for XML query processing with an indexing strategy. Internal data structures are built based on the data type definitions (DTD) of the XML documents. The internal data structures are used to store the data extracted from XML documents as well as storing indexes to the data items. The indexes keep track of the elements’ paths information. The system parses user’s query and uses the indexes to retrieve the data items that satisfy the conditions specified in the query.

1. Introduction

Indexing is essential to access records from large data sets. It is a well-developed technology in databases and other information retrieval applications. However, indexing semi-structured data, such as XML, is yet to mature and still being studied by many researchers. XML data sets, unlike traditional relational databases, do not have well-defined and fixed schema known in advance.

According to Robert Luk, et. al. [5], there are three broad approaches for XML document retrieval and indexing: database-oriented approach, information retrieval-oriented approach, and hybrid approach. The database-oriented approach is to actually store data in relational databases but present in XML rendered XSL stylesheet format. The information retrieval-oriented approach directly applies indexing to the retrieval of XML documents that are considered to be text documents. Index can be built on tags as if the tags were index terms. Particular words in certain tags may be treated differently based on the nesting-relationships between the tags. The hybrid approach combines some popular techniques and often creates XPath indexes for search. Sometimes multi-level indexes are built.

We have developed an XML query processing system using a hybrid indexing approach. The system is called XIQS (XML Indexing and Query System) that consists of three basic modules: graphical user interface, index building, and query processing. The DTD (Document Type Definition) of an XML document is parsed and the information about the structure of the document is extracted to build internal data structures for the data items and indexes. User’s query is parsed and checked for validity. Then the data satisfying the query conditions are retrieved using the indexes and the results are displayed in the GUI. The system also uses DOM and SAX to process the same query, just for performance comparisons with the indexing approach.

2. Related Work

In recent years, researchers and developers have developed many indexing schemes for XML data. Each of these schemes and methods is aimed to improve the performance of query processing by creating indexes at various processing stages. The indexing strategy proposed by J. McHugh, et. al. [1, 6] at Stanford University as part of the Lore project presented a general framework for index values in terms of automatic type coercion. Rizzolo and Mendelzon [8] of the University of Toronto proposed an indexing method that synthesizes ideas from object-oriented path indexes and extends to semi-structured realm of XML data. A 3-D bitmap indexing method for XML documents was proposed by Yoon, Raghavan and Chakilam [7]. Li and Moon [4] developed a novel approach for indexing and querying XML data for path expressions based on a tree-node numbering strategy and decomposition of the user’s query. One of the difficulties of building indexes for any data is the update of data items. To overcome this difficulty, one approach is to identify the index entries that are affected by the data change and update only those entries. That is, to reduce the number of updates to the minimum. Kha, Yoshikawa, and Uemura proposed an indexing method [3] that records the “relative region coordinates” (RRC) of elements in an XML document to solve this problem. Egnor and Lord of XYZFind Cooperation [2] suggested several ways to improve precision and recall for semantically structured XML document retrieval. They use an index structure that is a modification of the classic “inverted index.” Schlieder and Meuss [9] developed a retrieval technique that adopts the similarity measure of the vector space model, incorporates the XML document structure, and supports structured queries. The query model is based on tree matching, where XML documents and queries are labeled trees. Their approach allows partial matching of the query.

These are only some of the recent work in the area of indexing and retrieving XML data. This is still a field of great research interest and we would like to develop our own index engine as a basic platform for further research. The XIQS is the prototype of the system we have developed to support such research efforts.
3. System Overview of XIQS

The overall structure of XIQS is shown in Figure 1.

![System structure of XIQS](image)

Figure 1. System structure of XIQS

The graphic user interface provides a user-friendly environment for the user to browse the file directories, open files, select element types, and issue queries. DTD trees and query results are also displayed in the interface. It uses a Java component to render the XML query output in a tabular format.

When the user selects a DTD for which XML documents are to be retrieved, the DTD is parsed and the metadata about the XML documents are extracted and stored in internal data structures. The metadata will be used for validity check and query processing.

Indexes are created on each of the elements in the XML document that have values, either in the form of attribute(s) or leaf node. The indexes are stored in the file system and brought into memory as needed. Parent-child relationship is maintained by having some information about the children in the parent object. Metadata of the XML documents is used to check for validity of the user’s query and the index is employed to retrieve data that meet the query’s conditions. The system allows the user to choose one, two, or all of the three retrieving methods: DOM, SAX, or indexing, for the purpose of performance comparison.

4. Data Structures and Metadata

XIQS uses the tree concept of DOM and uses SAX to create indexes. It has its own set of data structures and access methods to store indexes, process query, and retrieve data. Index is created for each of the leaf and attribute nodes that have a value associated with it. These nodes on which indexes are created will have a unique path from the root of the XML document.

4.1. Data Structures

There are several data structures for storing metadata information as well as data from the XML documents. Information about queries is also recorded and saved. The main data structures are.

- **Element**: This data structure contains information about an XML element, including names and paths of the element and its attributes, names of files containing the indexes, etc.
- **Index**: This class contains a Vector of objects that store Element or Attribute values. Each vector of objects holds all the values for a unique element path in the XML document. Each vector represents an index data structure, which is eventually stored in a file. A Hash table maintains the mapping of the “indexed element” and “filename” in which the vector representing indexes is stored.
- **Attribute**: The attribute’s name, its value and its hierarchy in the XML document tree are recorded. Information about the immediate child nodes is also stored.
- **Query**: All the information of every element in the query is stored in a QueryElement object. Eventually, query is in the form of a vector of QueryElements.

4.2. Parsing DTD

To process a query, we need to know the structure of XML document being queried, and this could be obtained from the associated DTD. Thus, a DTD needs to be parsed to extract all the information about the XML document. The outline of the algorithm looks like this:

1. Read the DTD and create a vector containing all elements in the DTD.
2. The root is identified from the list.
3. A vector of objects is created wherein each object represents a unique path from the root.
4. Finally an object that represents the DTD in the form of a tree is created and the User Interface displays the tree for convenience of querying.

5. Creating Index

An index is created for every element in the XML document that has a value. The event-driven SAX parser is used to traverse through the XML document to create indexes. Indexes and the related information are stored in the file system as serialized java objects. Below are the methods that XIQS uses to create indexes and related data structures for each event.

**startElement(String name, AttributeList amap)**

The parameters specify the name of the element whose starting tag has been encountered and its attributes.

The system maintains a vector, XMLTags that contains the current path from the root to the element. The name of the element is added to the vector in startElement and removed from the vector in endElement. Similarly, an
AttributeData object is created for each attribute, and a set of AttributeData objects is created for every unique element path that has attributes. Each AttributeData object stores the names and values of the attributes and the path of the element having the attributes, as well as the information about its children, which is filled in at a later point when a child of the element is processed. Here is the algorithm handling the startElement event:

1. Add element name to vector XMLTags to maintain current path.
2. If (Element has attribute) create AttributeData object to store name, value, path.
3. If (Element has parent Element with attribute) InsertChildInfo into appropriate parent.
4. Add AttributeData object in the AttributeData vector.
5. Add the AttributeData vector in the AttributeObjects vector.

characters(char[] ch, int start, int length)

This method gets the data or text for the leaf node element. The data is added to the vector holding the index values of the element. This node could also be a child of some element with attribute. Hence the ChildrenInfo object of the appropriate parent attribute data object will also be updated. The algorithm is:

1. Get text data for the element, which is leaf.
2. Insert ChildrenInfo into parent AttributeData object.
3. Get data type of the leaf element.
4. If (Data vector for the leaf element exists) Add text data to vector.
5. Else
  Create new data vector for the element,
  Add text data.

endDocument()

This method receives notification when end of XML document tag is encountered. The indexes for the elements with attribute(s) and indexes of leaf elements are stored in files of the system in a specific directory. Mapping of element paths and the corresponding index file names along with the mapping of the element paths and their corresponding data type are stored in the file system.

6. Query Processing

When a user’s query, in which each value item is in the form of path expression, is submitted, XIQS parses it and checks its validity. The system then decomposes the query into basic parts and uses the indexes to retrieve the data that satisfy the query’s conditions.

6.1. Parsing Query and Validity Check

The query is decomposed into individual tokens that are categorized for validity check. The query can essentially be broken down to four parts:

- XML filename: It is stored as a String.
- Result Paths: They are stored in a vector of objects of type QueryElement. Result paths are paths of elements whose values have to be displayed upon query execution.
- Conditions: They are stored in a vector of objects of type Conditions. Each condition consists of left operand, operator, and right operand. Left operand is stored as QueryElement and represents a unique element path in the XML tree. Right operand can either represent a unique element path in the XML tree or can be a constant value.
- Joins: They are stored in a vector of Strings.

All the above information is stored in a ParseResults object. While the query is parsed, the system also checks for completeness. Basic requirements of a complete query are: (a) It should start with SELECT; (b) It should have the name of XML document that is being queried; and (c) It should have at least one result element path.

The validity check process is to ensure the correctness of the query, including the following:

- Each of the result element paths (field list) is correct and exists.
- If abbreviated element path is specified in the query, it should be a unique abbreviated element path.
- In condition expression, the data types of both operands should match.
- The operand paths in a condition expression should exist and be correct.
- The operator in a condition expression should be valid.

If an error is detected in the query expression, that portion of the query is highlighted and a warning is generated so the user can make corrections.

6.2. Data Retrieval (Query Execution)

The user’s query is executed and a Vector of Hash tables is created that stores the final results to be displayed. The execution steps are shown below.

1. Gets the parsed query information
2. Processes condition expressions if any
3. Processes joins if any
4. Retrieves values of result path(s) right away if there are no condition expressions
5. Stores result information in a vector of Hash tables

6.3. Performance of XIQS vs. DOM and SAX

Because XIQS builds indexes based on the metadata extracted from DTD, the existence of a DTD file for each XML document is mandatory. We have tested many queries on several XML documents using the proposed XIQS indexing technique as well as the traditional DOM and
SAX methods. The results show that the XIQS is much faster than DOM and SAX in most cases, as expected. XIQS is most efficient when the query deals mostly with leaf nodes. XIQS runs slower when an XML document involved in a query is processed for the first time, due to the time spent on building the indexes; or when the XML document has values mostly in the form of attributes, as indexes for attributes are a lot more complex than those for leaf nodes. If several consecutively executed user’s queries deal with the same XML document, the XIQS runs faster in general, depending on whether the data for the result element paths are already present in the memory or not.

The experimental results are for queries executed on an Intel Pentium-3 (850MHz processor, 128 MB RAM) under Windows operating system. Here we show the results of queries on three XML documents.

**XML document 1:**

<table>
<thead>
<tr>
<th>Query</th>
<th>XIQS Index</th>
<th>SAX</th>
<th>DOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0-110</td>
<td>110-220</td>
<td>110-160</td>
</tr>
<tr>
<td>2</td>
<td>0-50</td>
<td>160-220</td>
<td>50-110</td>
</tr>
<tr>
<td>3</td>
<td>50-110</td>
<td>170-220</td>
<td>110</td>
</tr>
<tr>
<td>4</td>
<td>0-50</td>
<td>110</td>
<td>50-110</td>
</tr>
<tr>
<td>5</td>
<td>0-110</td>
<td>60-110</td>
<td>50-110</td>
</tr>
</tbody>
</table>

**XML document 2:**

<table>
<thead>
<tr>
<th>Query</th>
<th>XIQS Index</th>
<th>SAX</th>
<th>DOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>380-390</td>
<td>270-710</td>
</tr>
<tr>
<td>2</td>
<td>0-110</td>
<td>330-380</td>
<td>220-440</td>
</tr>
<tr>
<td>3</td>
<td>0-380</td>
<td>770</td>
<td>280-380</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>380-440</td>
<td>280</td>
</tr>
</tbody>
</table>

**XML document 3:**

<table>
<thead>
<tr>
<th>Query</th>
<th>XIQS Index</th>
<th>SAX</th>
<th>DOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50-60</td>
<td>220-280</td>
<td>160-220</td>
</tr>
<tr>
<td>2</td>
<td>760-1150</td>
<td>220-440</td>
<td>220-330</td>
</tr>
<tr>
<td>3</td>
<td>4510-4670</td>
<td>600-660</td>
<td>390-440</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>160</td>
<td>220</td>
</tr>
</tbody>
</table>

As we mentioned before that the processing of conditions that involve attributes is complex and time consuming using indexing, the example 3 (that involves complex attributes) above shows that indexing method is much slower.

7. Summary and Future Work

We have presented in this paper a design of an XML query system that uses indexing mechanism to facilitate the query performance. Our goal is to build a model and an indexing engine that will enable us to further study various issues related to retrieving data from XML documents. The implemented indexing technique does not have cumbersome tree traversals as in DOM. Insufficient memory for very large XML documents is not an issue in the indexing technique as the data is brought to memory only when needed. Index data for a particular element is stored in a vector that makes access quick and easy.

Our experiments showed that the performance using the indexing method of XIQS is in general much better than DOM and SAX, particularly for querying those XML documents that were visited before, and for XML documents where data is mostly stored in the leaf nodes. We are currently working on indexing models for attributes to improve the execution time.

Bibliography


