A High Performance XML Querying Architecture

Fangju Wang

Hui Shen

A High Performance XML Querying Architecture

Fangju Wang, Hui Shen

Department of Computing and Information Science, University of Guelph
Guelph, Ontario, Canada, N1G 2W1

ABSTRACT

Data exchange on the Internet plays an essential role in electronic business (e-business). A recent trend in e-business is to create distributed databases to facilitate data exchange. In most cases, the distributed databases are developed by integrating existing systems, which may be in different database models, and on different hardware and/or software platforms. Heterogeneity may cause many difficulties. A solution to the difficulties is XML (the Extensible Markup Language). XML is becoming the dominant language for exchanging data on the Internet. To develop XML systems for practical applications, developers have to addresses the performance issues. In this paper, we describe a new XML querying architecture that can be used to build high performance systems. Experiments indicate that the architecture performs better than Oracle XML DB, which is one of the most commonly used commercial DBMSs for XML.

Keywords: XML, database, index, query processing

1. INTRODUCTION

In recent years, the needs for Internet data exchange between organizations have grown rapidly, especially in electronic businiss (e-business). For example, in e-business of B2B (business to business), companies may offer customers and partners to access their databases for information about their products and services. Brokers or consultants may need to process data from databases of many companies for best satisfying client requirements. There has been a trend in e-business to integrate existing stand-alone databases into distributed databases for facilitating data exchange.

XML (Extensible Markup Language) is becoming a major markup language for data exchange on the Internet. The language is especially useful in developing heterogeneous distributed databases, in which databases in different models, and/or on different kinds of platforms are combined into integrated systems. With XML, data from heterogeneous sources can be encoded as XML documents, transmitted on the Internet, and processed together. It has been widely accepted that XML is an indispensable tool in e-business, and that e-business cannot develop well without it.

Performance is a key issue in building a successful XML-based database. The querying techniques for the traditional relational databases were developed to manipulate record sets. High performance of the techniques is achieved by efficient read/write, search, and partitioning of records. However, XML documents are structured as hierarchical trees of nodes, which are essentially different from the table and record structures of the relational data. The querying techniques developed for relational data cannot be directly adopted for XML data. The special features of XML data require new techniques for querying XML data. Currently, research on querying techniques for XML data, including indexing and querying processing, is still at the preliminary stage. At the time of this writing, there have been no broadly accepted querying techniques XML databases.

In this paper, we describe a new querying architecture, which consists of three indexes, a block parser, and two query processing alsorithms. Experiments indicate that the architecture may perform better than Oracle XML DB, one of the most commonly used commercial DBMSs for XML data.

2. RELATED WORK

The related work includes XML data indexing and query processing for XML databases. Most papers about XML indexes focus on building path indexes on XML documents. In [3], an adaptive path index, APEX, is proposed to keep all paths to improve query performance. Ashraf and co-workers proposed two techniques: path tree and Markov tables [1]. They build a path tree to ensure that it fits in memory by deleting low-frequency nodes. Path information is contained in the Markov table.

In the work by Li [6], the index structure is composed of three major components: element index, attribute index and structure index, all of those indexes are based on a numbering scheme. For most index structures for XML data proposed so far, update is a problem because XML element’s coordinates are expressed by absolute values. In [5], Kha proposed an indexing structure based on the relative region coordinate that can effectively deal with the update problem.

The existing indexing techniques have major disadvantages. The most severe ones are that the sizes of indexes are very big and the time for building indexes for large XML documents may be very long. For example, in Lore, the total size of the indexes on an XML document of 465 megabytes is 2.2 times larger than the original file size and the time for building the indexes is about 20460 seconds (5.68 hours) [8].

Much of the work for XML query processing is based on graph theories. Shasha, and co-workers survey both algorithms and applications of tree and graph searching [10]. Prakash Ramanan develops O(n^2) and O(n^4) algorithms based on the concept of graph simulation to minimize tree pattern queries (TPQs) [9]. In the work by Fernandez and Suciu [4], two optimization techniques for queries with regular path expression, both replying on graph schemas, are proposed to restrict search on fragments of graphs and to find the most suitable optimization on all regular queries.

Structural join operations are central to evaluate queries against XML data. In [2], Bruno proposed a holistic twig join algorithm, “TwigStack”, for matching an XML query twig pattern. McHugh and Widom created a set of techniques to facilitate XML query processing in the Lore system designed specially for semi-structured data [7].

Currently, the research on querying XML documents is still at the preliminary stage. Some query techniques cannot support very large XML documents [9]. Some have too complex algorithms [9][7]. Some have to work with special indexing systems but the indexing systems are not optimal [4]. Some techniques have to rely on the underlying relational database structures but the join techniques developed for relational data cannot be directly adopted to join XML data.

3. THE INDEXING SYSTEM

As mentioned before, the index system in our approach consists of three indexes. They are path index, element index and block index.

3.1 Path index

The path index contains a set of paths extracted by parsing the schema of the XML document. Every path from the root to a node in the schema tree are identified by a unique number called *PID* (Path ID). Since every element in a document is associated with a node in the schema tree, the PID assigned to the schema node can be assigned to the class of elements in the document associated to the schema node. The path index can be used to find the PID of a given path.

3.2 Element index

The element index is built for finding all the elements associated with a given path. A pair of numbers is assigned to each element’s start tag and end tag, which are called StartEID and EndEID respectively. The StartEIDs and EndEIDs are sequential numbers and generated by preorder traversal on the document tree. An ID value is the same as the count of element tags in the whole document. StartEID is the number assigned to the start tag of an element and EndEID is the number assigned to the end tag. For an element containing string value only, EndEID is the same value as StartEID.

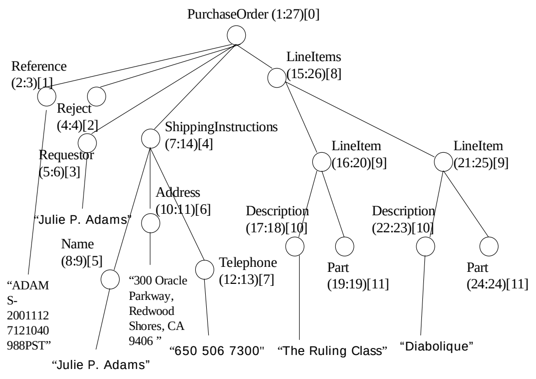

Figure 1.The element numbering strategy

Figure 1 shows an XML document tree and the StartEID and EndEID pairs assigned to the elements. Since every element in a document is associated with a node in its schema tree, the PID assigned to a schema node can be assigned to the corresponding elements in the document. We can search the path in the path index to get its PID, and assign the PID to the element. PID is included in “[]”on every element in the Figure. The StartEID:EndEID pairs with the same PID are stored into a file. The collection of such files is called the element index.

3.3 Block index

The block index is used for identifying the exact file block on disk in which a given element is stored, so that the block can be directly accessed without sequential search. The index contains the StartEID and EndEID pair and PID of the first and last element in every block. A block index is built by recording the information about the tags of the three elements in each block:

1. The first “element with its start tag” in the block.
2. The first “element with its end tag” in the block.
3. The last “element with its start tag or its end tag” in the block.

A tag in XML is always associated with an element. Since an element in the document has an associated PID in the path index, the tags of that element can also be associated with that element’s PID. In this paper, a tag’s PID refers to its associated element’s PID. A tag’s EID refers to its associated element’s StartEID or EndEID.

The block index plays an important role in achieving the high performance. By comparing a given element’s StartEID and ENDEID with the blocks’ tag Ids, the block containing the element can be decided and directly read from disk. Then the information about the three elements listed above can be used for block parsing, which will be described in the next section.

4. QUERY PROCESSING ALGORITHMS

In this section, we describe the algorithms based on the indexes for achieving high querying performance.

4.1 Block identification algorithm

The block identification algorithm is used to find the disk blocks containing a given element. The algorithm compares the element's StartEID and EndEID pair with each block's tag information in the block index. In this algorithm, variable *StartBlock* is used to represent the block containing the tag with StartEID and variable *EndBlock* is used to represent the block containing the tag with EndEID. Either StartBlock or EndBlock is the sequential block number assigned to a block in the block index. The input of this algorithm is the StartEID and EndEID pair of the element to be found, the output is the StartBlock and EndBlock pair of the block(s) to read. The content of the element crosses over the blocks between StartBlock and EndBlock.

4.2 Block parser

After the block or blocks containing a given element are read from the disk, we need a special parser to parse the content in the blocks to get the element. This parser is the *block parser*. Along with the block index, the block parser allows an element to be retrieved quickly.
There are two popular XML parsers: SAX and DOM. Both have to scan the entire document from the beginning to end. Scanning XML documents is usually costly. The block parser only parses the contents of the blocks of interests instead of scanning the whole document. It thus helps minimize the I/O costs in retrieving elements.

In the block parser, an *EID counter* is defined to trace the element ID assigned to every element's tag and a s*tep stack* is defined to trace the element's name in the block. The block parser initializes the EID counter and the step stack by the tag's information recorded in block index.
The blocks to be parsed are a sequence of blocks between StartBlock and EndBlock. The parsing is started from the first character in the first of those blocks. When the block parser reads a start tag, it pushes the element's name in that tag as a step onto the top of the step stack. When the block parser reads an end tag, it pops up the step from the top of the step stack. There is an exception: when the first tag is a start tag, block parser ignores it and does nothing.

A path, here called s*tack path*, can be built by connecting the steps from the bottom to the top in the step stack. Every time after pushing an element's name to the step stack, the block parser builds a stack path and looks up for PID in path index for the stack path. Here, the PID is called s*tack PID*. If the stack PID is among the PIDs of the source nodes in the query trees, the block parser stores the content between the current start tag and its matching end tag into a buffer in main memory or to temporary files on disk.

4.3 Single-root query algorithm

We categorize XML queries into the *single-root queries* and *multi-root queries* and apply different algorithms to process them. In the following discussion, the queries are represented in the Xquery language. A query has a For clause, a WHERE clause, and a RETURN clause.

In a single-root query, there is only one variable defined in the FOR clause that can serve as the root of all the paths referenced in the query. *Source elements*, which are elements referenced in the WHERE clause, must be processed in the scope of root elements. Source elements in different root element scopes cannot be processed together. The result of the operation can be generated within the scope of every root element.

First of all, the PID assigned to the root node is used to obtain the first StartEID and EndEID pair from the element index. The first root element's scope can be built by the StartEID and EndEID pair. Then, the PID assigned to every source element is used to read the corresponding element index. By an algorithm for determining the "ancestor-descendent" relationships, every source element's StartEID and EndEID pair is compared with the root element's scope and then all source elements' StartEID and EndEID pairs in the current root query element's scope are decided.

Through the "Block Identification Algorithm", the exact blocks containing a desired source element in the WHERE clause can be identified. The content of these blocks can be read and parsed by the block parser to fetch the content of that element to a buffer. When all the source elements in the WHERE clause are fetched, the query condition is evaluated. When a set of source elements satisfies the query condition, the blocks containing the associated elements in the RETURN clause are identified. The content of the blocks are parsed and values of those elements are output as the result immediately. Figure 2 is the diagram illustrating this algorithm.

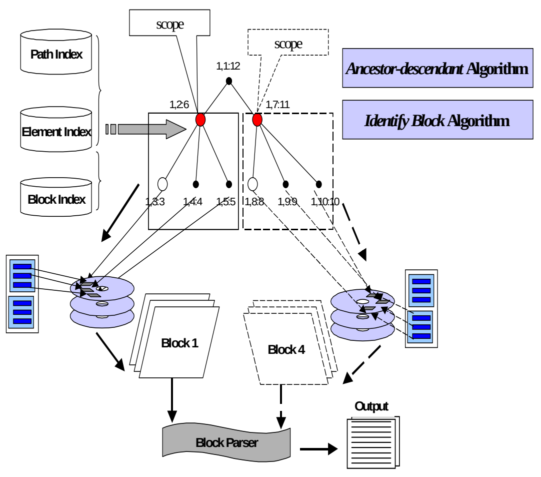

Figure 2. Single-root query algorithm

4.4 Multi-root query algorithm

In a multi-root query, there are more than one variable defined in the FOR clause that can serve as roots of the paths referenced in the WHERE and RETURN clauses. The multi-root query algorithm is more complicated than the single-root query algorithm. Roughly a multi-root query is similar to a join query in the relational data model. A join operation is related to two root element classes. In the following, "A" refers to the first of the two root element classes and "B" refers to the second root element classes. A source element class in the WHERE clause is called *WHERE element class*. The element in a WHERE element class is called a *WHERE element*. A WHERE element class is always associated with root element class "A" or "B". The join operation is performed by comparing a WHERE element within the scope of an element of "A" with all the WHERE elements within the scopes of all the elements in "B".

We can use an element class's PID to get its element index. From A's element index, we can get the scope of each element by its StartEID and EndEID pair. We can get a WHERE element with a StartEID and EndEID pair within this scope through the algorithm for determining the ancestor-descendent relationship. Then the blocks containing the WHERE element can be identified by the Block Identification Algorithm. The WHERE element can be fetched by using the block parser. And then the fetched WHERE element and this scope are stored together into a value set.

The procedure is conducted on all the pairs of WHERE elements. At a point of time we can get two sets of WHERE elements. The first contains the WHERE elements within the scope of an element in "A". The second contains the WHERE elements within the scope of an element in "B". We execute the query operation on these two sets of WHERE elements. The join operation would be executed by comparing every WHERE element in the first set with all WHERE elements in the second set.

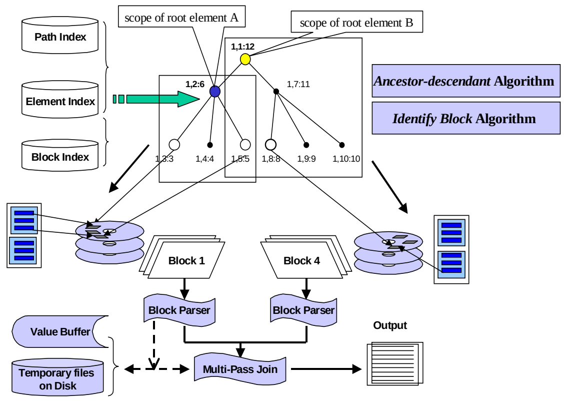

Figure 3. Multi-root query algorithm

When the join condition is satisfied by a pair of WHERE elements, the pair of the corresponding root element's scopes is recorded as an *intermediate root scope pair*. The two scopes in an intermediate root scope pair represent the root element in "A" and in "B" respectively, and hereafter called *intermediate scope A* and *intermediate scope B* respectively. After the execution of the join operation, we can get a set of intermediate root scope pairs.

Every source element class in the RETURN clause (here, called *RETURN element class*) is associated with a query node in the query tree with its associated root query element class "A" or "B". The element in a RETURN element class is called *RETURN element.* Only the RETURN element in either an intermediate scope A or B is *valid* for the output. Figure 3 shows the multi-root query algorithm.

5. EXPERIMENTS AND RESULTS

In this section, we describe the experiments and present experimental results.

5.1 Test environment

The computer used for the experiments is a desktop station with a 2.0 GHz Intel Celeron IV processor, 512 MB RAM and a 120 GB Seagate hard disk. The querying architecture is implemented in the C language on Linux 7.3. For comparison, the same queries are executed in Oracle XML DB, which is installed on Windows 2000. Both the Windows and Linux operating systems are on the same machine.

The test data are from Oracle. They are XML documents of sizes of about 3, 8, 10, 30, 60, 100, 300, and 600 megabytes, on a schema of "purchasing orders". For each document, we examine the total size of indexes and the time for creating indexes in both our system and Oracle XML DB. A set of queries are applied to the data. The queries are used to perform operations of selection, equal join, θ-join on the data. The time required by both systems are recorded and compared.

5.2 Experimental results

5.2.1 Building indexes

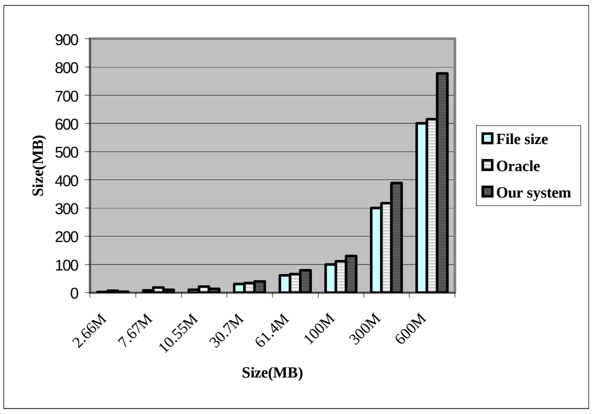

Figure 4 Space used by the two systems.

A path index is a memory structure that is generated by parsing the schema. Generally the size of a schema is very small, so we ignore the time for parsing a schema to create the path index. The time spent on building the other two indexes is significantly affected by the time for parsing the document and the time for writing element index and block index to disk.

Figures 4 and 5 gives the comparison of the space usage in Oracle and our system, and the time required by the two system to organize data and create indexes. Our system needs 20% to 40% of the time by Oracle XML DB and more space when data sizes are big.

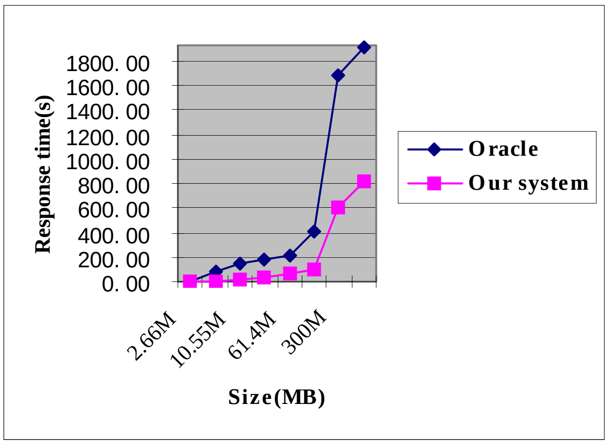

Figure 5. Time required by the two systems.

5.2.2 A query of selection

The following query is designed to select an element and retrieve its parent as well as all the children of the parent element. This is a single-root query.

```
For $c in document("PurchaseOrder.XML")/PurchaseOrder/
Where $c/Reference="ADAMS-20011127121040988PST"
Return $c
```

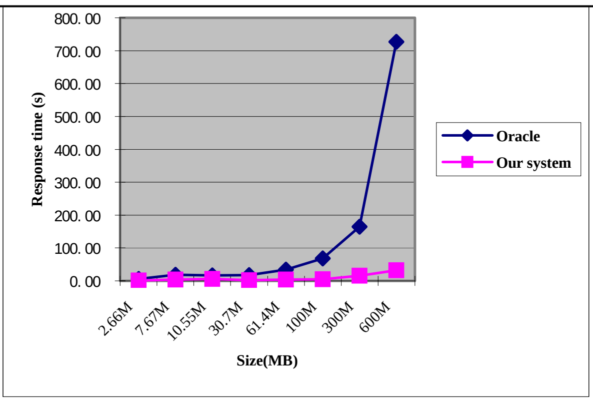

Figure 5.Comparison of the time for executing the selection query.

Figure 5 illustrates the response time for evaluating the selection query with XML documents of different sizes in both Oracle XML DB and our system. The results indicates our system is faster than Oracle XML DB when executing such a query.

5.2.3 A query of document join

The following is a query for joining two XML documents. The WHERE clause includes an equality condition. Element pairs satisfying the condition are selected and the number of the elements in one of the classes are returned as query results.

```
For $c in document("PurchaseOrder1.XML") /Purchase
Order/,
$d in document("PurchaseOrder2.XML")/Purchase
Order/ShippingInstructions
Where $c/Requestor=$d/Name
Return  DISTINCT({$c/Requestor})
```

Figure 6 shows the experimental results in testing this query. The time required by our system is basically proportional to the document size. For most of the documents, our system performs better than Oracle XML DB. Since we have no knowledge about how exactly Oracle handles the data, we are unable to explain the behavior of Oracle.

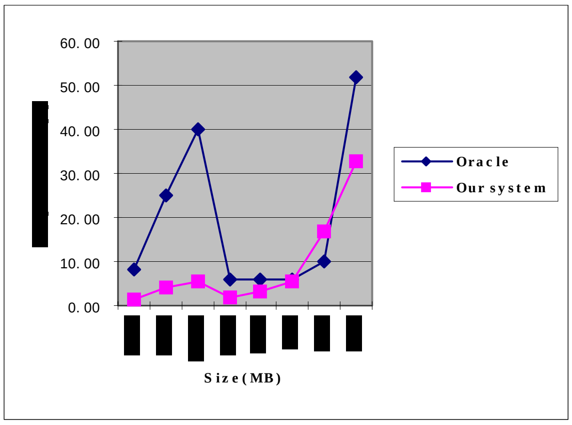

Figure 6. Comparison of the required for executing the join query.

6. CONCLUSION

In this research, we develop a novel architecture for querying XML data. The core of this architecture includes the block index and block parser, which overcome disadvantages of the existing general purpose indexing systems: large sizes and long creation time. Based on the indexing system, a set of algorithms can achieve high performance in evaluating queries. Experiments indicate that the system implementing the architecture requires less time to create the indexes than the time by Oracle XML DB to load the same files. Our system is 2 to 12 times faster than Oracle XML DB. For most of the test data, our system is faster than Oracle XML DB in executing the same queries. Our system requires only 0.5 % to 50% of the time required by Oracle XML DB to evaluate the same test queries.

REFERENCES

1. A. Aboulnaga, A. R. Alameldeen and J.F. Naughton, 2001, “Estimating the selectivity of XML Path Expressions for Internet Scale Applications”, *Proceedings of the 27th VLDB Conference*, Roma, Italy.
2. N. Bruno, N. Koudas and D. Srivastava, 2002, “Holistic Twig Joins: Optimal XML Pattern Matching”, *ACM SIGMOD 2002*, Madison, Wisconsin, USA.
3. C. W. Chung, J. K. Min and K. Shim, 2002, “Apex: An Adaptive path index for XML data”, *ACM SIGMOD* ‘2002 June 4-6, Madison, Wisconsin, USA.
4. M. Fernandez and D. Suciu, 2002, “Optimizing Regular Path Expressions Using Graph schemas”, *Proceedings of IEEE 14th International Conference on Data Engineering.*
5. D. D. Kha, M. Yoshikawa and S. Uemura, 2001, “An XML Indexing Structure with Relative Region Coordinate”, *17th International Conference on Data Engineering.*
6. Q. Li and B. Moon, 2001, “Indexing and Querying XML Data for Regular Path Expressions”, *Proceedings of the 27th VLDB Conference*, Roma, Italy.
7. J. McHugh and J. Widom, 1999, “Query Optimization for XML”, *Proceedings of 25th VLDB Conference*, pp315-326, Edinburgh, Scotland.
8. L. K. Poola and J. R. Haritsa, 2001, “SphinX: Schema-concious XML Indexing”, Database Systems Laboratory, Dept. of Computer science & Automation, Indian Institute of Science, Bangalore.
9. P. Ramanan, 2002, “Efficient Algorithms for Minimizing Tree Pattern Queries”, *In ACM SIGMOD 2002 June 4-6*, Madison, Wisconsin, USA.
10. D. Shasha, J. T.L. Wang and R. Giugno, 2002, “Algorithmics and Applications of Tree and Graph Searching”, *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems'*, Madison, Wisconsin.