ISSN (Print) : 0974-6846
Indian Journal of Science and Technology, *Vol 9(31), DOI: 10.17485/ijst/2016/v9i31/92731, August 2016* ISSN (Online) : 0974-5645

An Improved Indexing Method for Xpath Queries

Hoang Do Thanh Tung[1*] and Dinh Duc Luong[2]

[1]Institute of Information and Technology, Vietnam Academy of Science and Technology (VAST), Hanoi, Vietnam; tunghdt@ioit.ac.vn
[2]Food Industrial College, Phu Tho, Vietnam

Abstract

Today, the XML is used as data storage for complex data models like bioinformatics information. A bioinformatics system deals with large data sets and complex queries. Thus, it is necessary to have accessing methods for XML data. XPath is a method to quickly locate any information that we need in an XML (tree) data starting from the context node in a root node to subtrees. In this paper, we propose a system model to store XML data more efficiently and also an improved indexing method to support Xpath queries. In the system model, we integrated big data model with relational data model in order to get benefit from both of them. The new indexing method is an improvement of R-tree that helps Xpath queries run more efficiently in some axes. Our experiments showed that the proposed method gains better results for node queries compared to the R-tree in transformed XML data. Our method is intended to apply to phylogenetic queries of Treefam databases.

Keywords: Bioinformatics, Hadoop, Indexing, XML Data, Xpath Queries

1. Introduction

With such rapidly new applications, storing and querying XML data becomes strongly interested. Several XML query languages have been proposed such as Xpath and XQuery. The common feature of such languages is to use regular path expression to query XML data. This comes up with a new requirement for indexing and searching XML database because XML data models are basically in tree structure, where nodes represent elements, attributes and text data. Parent-child node pairs represent relationship between XML data components.

In XML documents, Xpath provides path-operators to describe traversals. Starting from the context node, an XPath query traverses its input document using a number of steps. A step's axis indicates which tree nodes are reachable from the context node. The XPath specification lists a family of 13 axes.

To speed up the processing of regular path expression query, it is important to be able to quickly determine all the relationships between any pair of nodes in the hierarchy of XML data such as ancestor-descendant, parent-child...

Currently, XML data has a major role in addressing some of the problems of the bioinformatics community because

- Bioinformatics data is a complex model with many different types, numerous relationships; new types of data emerge regularly and data are updated very frequently, accessed intensively and exchanged very often by researchers on the internet.
- Bioinformatics data consists of terabyte of granular objects. Managing (that is, modeling, storing and querying) this information is still big challenged. It has been recognized as a key component of today's genome/biology research. It is now starting to be a botlenect in many biological projects.

Additionally, one of the reasons we got interested in Xpath queries for XML data is Treefam. TreeFam (http://www.treefam.org) is a database of phylogenetic trees inferred from animal genomes, providing homology, orthology predictions together with the evolutionary history of the genes. It has orthology predictions and gene trees for 109 species in 15 736 families covering 2.2 million sequences.

From the above motivations, we come up with some proposals for this paper

- A system model that can deal with large size and complex model of bioinformatics data and also manage easily.
- A new indexing method is an improvement of R-tree

* *Author for correspondence*

that helps Xpath queries run more efficiently in some axes. This method helps to access deeper nodes directly such as children instead of descendants, parents instead of ancestors, sibling instead of following...

In the rest of this paper, we review some related works on continuous queries in section 2. Then, we present our proposed method in section 3. Section 4 shows our experimental results. We conclude our method in section 5.

2. Related Works

An Xpath query scanning the entire XML data will cause significant performance degradation. Indexing methods have been developed in recent years to overcome this issue. Different XML data indexing and query processing methods have been proposed to support Xpath queries. The paper[1,2] classified XML data indices into three categories.

- The first is path-based indices which group nodes in data trees by local similarity and have an index structure adjustable according to the query workload. The drawback of these indices is a huge index size because they store the forward and backward paths to establish a supportive layer. See[3,4].
- The second is sequence-based indices which evaluate queries by sequence matching after transforming both XML data and queries into sequences. However, their drawback is the occurrence of false positive that is caused by sequence matching instead of tree matching. See[5,6].
- The third is node-based indices which uses the pre-order values (or together with post-order values) of a node on the XML tree as coordinates of the node on two-dimension plane. The result is then used to determine the relationships between tree nodes.

The method of this paper focuses on the third type. The node-based indices can be further classified into the numbering-based index and multi-dimension-based index.

The numbering-based index relies on labeling XML nodes to represent the position of nodes within a document. Paper[7] shows the range-based sub-tree index, the regional- based sub-tree index and the prefix-based labeling. The range-based sub-tree index is used to determine the ancestor–descendant relationship between any pair of XML elements. Its drawback is that re-computing global order of nodes is requested whenever a node is inserted, see[8,9] and poor query performance. The regional-based sub-tree index uses pairs of (start position, end position) of the substrings of XML data counted from the start of the XML document on depth-first traversal. Its drawback is unable to handle frequent updates of XML data because it is assumed that the node positions are never changed once they are assigned, see[10]. The prefix-based index is to precisely indicate an ancestor of a node on its path. Its drawback is that when the depth of a tree increases, the size of the label increases quickly, see[11,12].

In the multi-dimension-based index in[13,14] maps every element node onto the two-dimension plane, using its pre-order rank on the x-axis and, its post-order rank on the y- axis. The context node and the four major axes of the XPath steps (descendant, ancestor, following, and preceding) divide the two- dimension space into four document regions, each corresponding to one major axis. Given a context node, the process of calculating one of its axes can be simplified as partitioning nodes on the two- dimension plane and retrieving the nodes falling into the region corresponding to the specific axis in a query. An index structure, XPath Accelerator, has been proposed to support the multi- dimension approach. By using a relational database system, it will benefit greatly if the database supports spatial indexing techniques such as R-tree[15]. The resulting set of nodes of the four major axes will be combined to support all path expression axes. Optimizations can be performed to further reduce the size of the document region in a query, though the pre-rank and post-rank in some cases of node insertions have to be re-calculated.

The drawback of multi-dimension-based index is inherently overlapping problem of the bounding rectangles that can reduce range-query performance. To limit use rang-queries, in this paper, we will propose an improved indexing method.

3. Proposed Method

3.1 A System Model

Currently, the number of XML data files are very large and complexity after the explosion of applications on Internet. XML data is semi-structure so it's very open and flexible in content of data. Their data schema is not as constant as that of traditional relational database. Therefore, before processing their data, they should be preprocessing in order to reduce data size as well as complexity. In

practical, bioinformatics data is the most evidence under form of XML data.

Data of DNA sequences is decoded by different biotechnology centers in the world different. One of difficulties for information managers is that data from the different centers can be stored under different formats. Moreover, the data in the various information centers such as NCBI, EBI, DDBJ are also be saved in different formats. To solve problems related to differences in data format, they need defined standard data formats for the purpose of storing and sharing data. FASTA and FASTQ are the standard data format, simple and important. This is the XML format for storing information on the DNA sequence.

The system model we propose is an inter-connection between Big data server (Hadoop) and SQL server which has R-tree Indexing engine used in our next proposal. In this model, the Hadoop is responsible to automatically collect XML data from the expected multi-sources through networks. There is a software component waiting and automatically extract necessary data, get them into relational tables and transmit the tables to SQL server. In turn, SQL server is responsible to store and index the data.

Figure 1. Inter-connection between Big data and SQL server.

In figure 1, Hadoop is responsible to store all XML data and SQL server is responsible to store data for indexing. The combination of these two systems provides balance between building systems and data usage. With the big data, semi-structured data is gathered and stored in Hadoop. However, the processing of large volumes of data directly will not be able to meet the requirements of performance. Thus, the key data will be extracted, transformed into a structured data and stored into the data warehouse of SQL server. In our experiment, we store XML data of gene sequences, such as maize, potato, cassava in Hadoop, and then separate the necessary information such as gene sequences. We put them on the data warehouse. Thanks to that, indexing process work becomes easier.

3.2 An improved Indexing Method

When designing an access method, we not only get interested in the nature of the data, but must also the types of queries, Xpath. Practically, the R-tree-based indexing method can solve most of those queries but not address the siblings as well as the complex queries with predicate especially with location parameters without filtering steps for final results. We need to find an optimal indexing structure that can afford all the axes. There are two goals for the proposal: we want to avoid the overlapping problem as much as possible by restricting the use of range queries and obtain query results without the immediate steps. Therefore, our proposed indexing structure can afford all kinds of structure-based query, especially focus on solving the predicate queries.

By following the way of Xpath Accelerator, in order to capture the particularities of XML data and queries, we propose a new access method called XR-tree. Basically, the design of XR-tree is similar to that of the R-tree index. However, we use a technique in terms of persistently keeping trajectories of siblings of a parent node in XML data on an XP-tree index.

3.3 Indexing Structure

We design XR-tree structure that that strictly preserves sibling trajectories. A trajectory keeps sibling nodes who are children of the same parent. Thus, the structure of the XR-tree is different from that of R-tree in that a leaf node contains only partial children of one parent. In other words, children of one parent are distributed over and connected through a set of leaf nodes. Each node in XML document after parsing is represented as an entry of 5 attributes. node(v)={pre(v), post(v), par(v), att(v), tag(v)}. In XR-tree, each leaf node entry is a set of (preorder, postorder, parent, attribute, tag). Non-leaf node entries are in the form (pointer, MBB) where pointer points to a child node and MBB is a conservative bounding rectangle. Leaf nodes are responsible to keep trajectories of actual XML data so that each leaf node additionally includes a set of pointers (previous-pointer, next-pointer, parent-pointer). To do that, we apply doubly linked lists to keep connections to previous XML sibling and following XML sibling. We also use one pointer to keep connection to the parent of XML children. In summary, we use 3 pointers in a leaf node to connect to previous sibling, next sibling, and parent of its entries. It benefits that we can quickly trace sibling relationships in XR-tree.

The drawback of the structure of XR-tree is that we must sacrifice space of leaf nodes to contain additional pointers and waste some empty entries if the number of XML children is sparse. In practical, we found that the number of children is much larger than that of ancestor. Thus, we get much benefit from query processing.

3.4 An Insertion Algorithm

To keep the sibling trajectory of the XML data. The insertion algorithm has to be modified to calculate additional pointers. We assume that new entries are inserted sequentially following the trace of traversal Pre-order or Post-order, we call the trajectory of the entries. To insert a new entry (a XML node), we first find the leaf node that contains its previous siblings or following siblings who is in the same trajectory. To do that, we start by traversing the tree from the root and jumping into every child node that overlaps with its parent's MBR by a algorithm, **newFindNode. newFindNode** will return leaf nodes that contain the previous (following) sibling XML nodes of the new entry. Secondly, we must choose the leaf node that contains the last XML siblings of the new entry. Finally, if the leaf node still has place for new entry, we will insert new entry to the leaf node. In case the leaf node is full, we will use **newSplitnode** algorithm in order to split the leaf node and re-set pointers.

3.5 Query Processing

The query algorithms of the XR-tree are nearly the same to that of R-tree. However, to take advantage of trajectory of siblings, we improve the algorithms in two types of queries, Node (point) query and Rang query. We will describe the two typical types of queries, point or node (for XML data) query and range query as following

3.5.1 Node Query

After reaching the leaf node containing the search XML entry in the XR-tree, the pointers of the XR-tree node allow us to trace XML sibling in connected nodes very easily. We have two possibilities: a sibling entry can be in the same leaf node or in another node. If it is in the same, finding it is trivial. If it is in another node, we simply follow the next (previous) pointer to the next (previous) leaf node. Similarly, we can use the pointer to find parent-child relationship of the entry.

3.5.2 Range Query

A range query is to look for XML nodes beyond sibling relationship of the search entry. A range query requires searching in a specific space instead of an XML entry. Normally, queries can be of the form "find all the descendant nodes from the context node" or "find all the children of the second node from the context node". The query result should be in a number.

Given a node f, a range query should fall into one of four disjoint regions of a plane: 1. the lower-right partition for descendants of f; 2. the upper-left partition for the ancestors of f; 3. the lower-left region for the nodes preceding f in document order; 4. the upper-right partition for the nodes following f in document order.

4. Experiments

We did two experiments, the first for the proposed system model and the second for our improved indexing method.

4.1 A Proposed System

In this experiment, we try to store and query XML data of bioinformatics in different storages of Disk Based – caching full RAM, Disk Based – caching 512 MB RAM, In Memory and Hive (Hadoop). Then we will tranform those semi-structure data into structure data stored in SQL server 2014 in form {pre(v), post(v), par(v), DNA of v}. All experiments are deployed in a computer.

The most important feature is the data query speed, we tested query speed of 1,163,440 records of XML data.

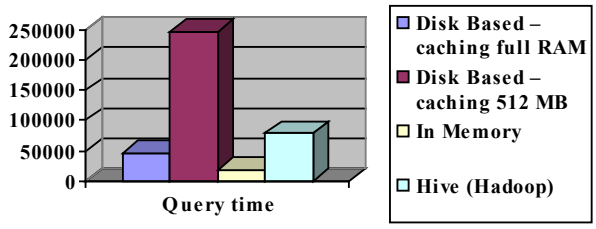

Figure 2. Querying XML data in different systems.

The speed of the Hive (Hadoop) is better than that of conventional drives, but is slower than that of caching on RAM or In-memory, see figure 2.

However, when we try a XML data that contains the DNA of corn with 3.1GB (approximately 63.9 million

information records), SQL Server got errors of storage but Hive (Hadoop) done directly and quickly. We then extract XML data into SQL server in form data table, see figure 3.

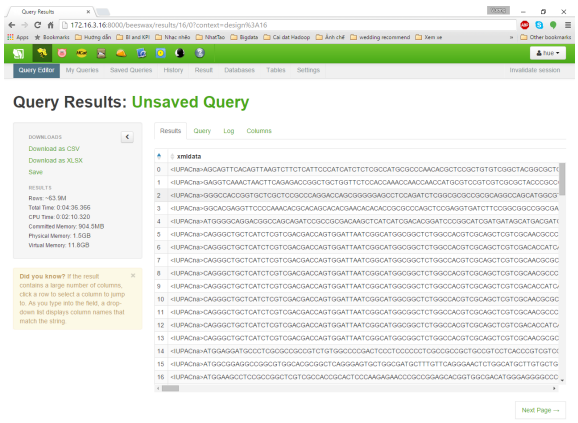

Figure 3. Gene data extracted from Hadoop.

This experiment proved that our system model works efficiently in storing XML data large and complex like genome data as well as speeding up extracted data in SQL server with In-memory. Additionally, it's clear that we can take advantage of R-tree to index and analyses the extracted data much easier and more quickly. In next experiment, we will test our new indexing method by R-tree on this extracted data.

4.2 An Improved Indexing Method

We use some XML data files with different sizes but not too big because we all run on a computer. In this experiment, the input is an entry or object of a XML file, our responsibility is to find all its relation. We took two kinds of query, node queries (**sibling preceding, sibling following, children**) and range queries (**ancestor, descendant, following, preceding**).

4.2.1 Node Queries

The Figure 3, 4 shows that XR-tree's performance is much better than that of R-tree. The reason is that to gain results, R-tree has to use a range query to scan all sibling or descendent nodes, and then filter out expected nodes. But XR-tree processes these queries by first only reaching a leaf node containing the object and then searching all it sibling, children nodes through pointers. This helps to avoid overlapping problem of R-tree. The larger XML data size is, the more overlapping an R-tree meets. It is reason why the performance of the R-tree rapidly declines when the data size increases.

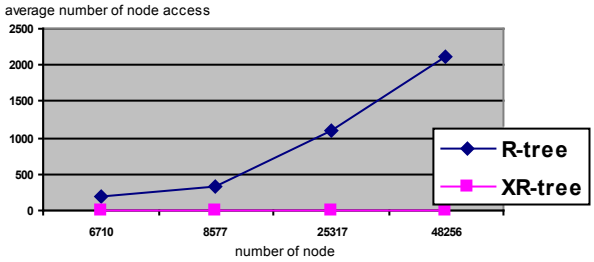

Figure 3. Sibling queries.

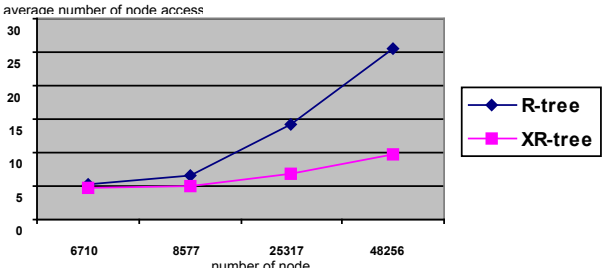

Figure 4. Children queries.

4.2.2 Range Queries

Range queries force both R-tree and XR-tree to search a lot of objects related to the object, all ancestors or descendants. Normally, they have to use rectangles with a quarter of a plane to search nodes of a certain axis. This generates many redundant steps because of overlapping problem and lead to more time-consuming in searching and processing data.

Figure 5 shows that XR-tree's performance is a little bit worse than that of R-tree except large size data. The reason is that we forced the sibling nodes of an object of XML data into some leaf nodes of R-tree. Certainly, that makes the indexing structure less optimal, resulting in raising overlapping problem. Despite of that, thanks to pointers (to parents) of XR-tree, XR-tree's performance is not much worse than that of R-tree even better when the amount of overlaps or data size becomes large enough.

Figure 6 shows that that XR-tree's performance is a little bit better than that of R-tree. Instead of scanning entire one of four disjoint regions on the plane, XR-tree only looks for children of a descendent node and then use pointers (to sibling nodes) to gain the rest.

Similar to Figure 5, Figure 7, 8 shows that XR-tree's performance is a little bit worse than that of R-tree. The reason is that a range queries forces to scan one of four disjoint regions that result in severely overlapping

problem.

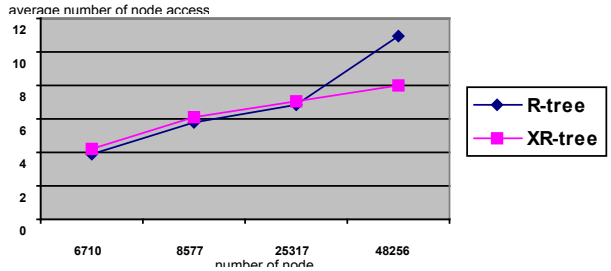

Figure 5. Ancestor queries.

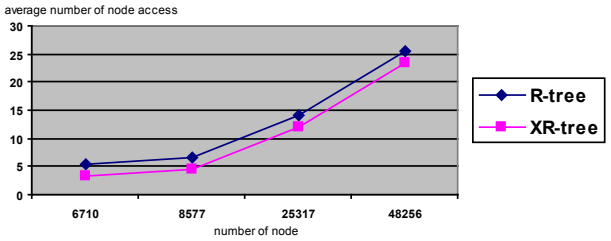

Figure 6. Descendant queries.

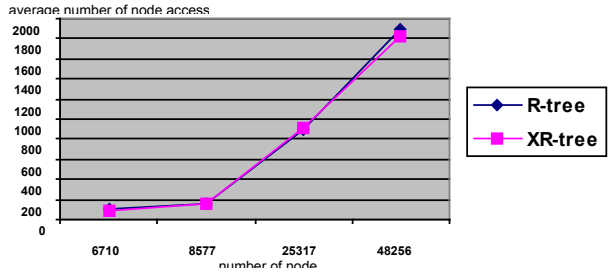

Figure 7. Experiment with following query.

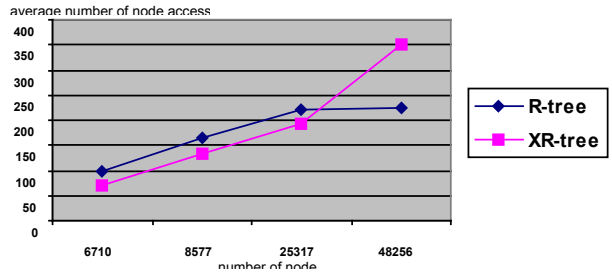

Figure 8. Preceding queries.

In summary, our proposed indexing methods get much better performance with node queries but nearly similar performance with range queries. Fortunately, node queries are much more practical than range queries because users rarely need all following, ancestors or descendant data of an object. Practically, sibling preceding, sibling following, children queries much benefit Treefam database to search and curate tree.

5. Conclusions

Desiring to manage the XML data and increase the efficiency of Xpath queries for bioinformatics data under XML form, we have proposed a model system for storage and increasingly efficient use of XML data, and also proposed an indexing method that is a modification of the R-tree. Experiments proved that our indexing method has better performance compared to that of R-tree with node queries.

One more advantage of the proposed method is that SQL server (or the similar) supports R-tree, this makes practical implementation of our approach more easily. With the proposed system, the conversion of unstructured XML data to a structured format managed by SQL Server will be more effective in indexing and analyzing data. But we do not give an assessment of that issue here.

In the future, we plan to apply as well as improve our method to bioinformatics data, especially problems of phylogenetic trees from animal genome databases.

6. Acknowledgment

This paper is supported by project VAST01.09/14-15, Vietnam Academy of Science and Technology (VAST), Hanoi, Vietnam.

7. References

1. Haw S, Lee C. Data Storage Practices and Query Processing in XML Databases: A Survey. International Journal of Knowledge-Based Systems, Elsevier. 2011; 1317–40.
2. Alghamdi NS, Rahayu W, Pardede E. Semantic-based Structural and Content indexing for the efficient retrieval of queries over large XML data repositories. Journal of Future Generation Computer Systems. 2014 Jul; 212–31.
3. Chung J, Min CW, Shim K. APEX: an adaptive path index for XML data. Proceedings of ACM SIGMOD. 2002; 121–32.
4. Han JY, Liang ZP, Qian G. A multiple-depth structural index for branching query. Journal of Information and Software Technology. 2006; 928–36.
5. Rao P, Moon B. PRIX: indexing and querying XML using

prufer sequences. Proceedings of ICDE, IEEE. 2004; 288–300.
6. Tatikonda S, Parthasarathy S, Goyder M. LCS–Trim: dynamic programming meets XML indexing and querying. Proceedings of the 33rd International Conference on Very Large Data Bases. 2007. p. 63–74.
7. Haw S, Lee C. Node labeling schemes in XML query optimization: a survey and trends. Journal of IETE Tech Rev. 2009; 88–100.
8. Dietz P. Maintaining order in a linked list. Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, ACM. 1982; 122–7.
9. Li Q, Moon B et al. Indexing and querying XML data for regular path expressions. Proceedings of the International Conference on Very Large Data Bases. 2001. p. 361–70.
10. Zhang C, Naughton J, DeWitt D, Luo Q, Lohman G. On supporting containment queries in relational database management systems. Journal of ACM SIGMOD Record, ACM. 2001; 425–36.
11. Tatarinov I, Viglas SD, Beyer K, Shanmugasundaram J, Shekita E, Zhang C. Storing and querying ordered XML using a relational database system. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02, ACM, New York. 2002. p. 204–15.
12. O'Neil P, O'Neil E, Pal S, Cseri I, Schaller G, Westbury N. Ordpaths: insert friendly XML node labels. Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD '04, ACM, New York. 2004. p. 903–8.
13. Grust T, Van Keulen M. Tree awareness for relational DBMS kernels: Staircase join. Journal of Lecture Notes in Computer Science. 2003; 231–45.
14. Grust T, Keulen MV, Teubner J. Accelerating XPath evaluation in any RDBMS. Journal of ACM Trans Database Syst. 2004; 91–131.
15. Guttman A. R-Trees: A dynamic index structure for spatial searching. Proceedings of SIGMOD, Boston, Massachusetts. 1984.