Structural Join in the 'XSQS' Native XML Database

Dong Li
School of Software Engineering, South China University of Technology, Guangzhou 510006, China
Email: cslidong@scut.edu.cn

Xiuyu Lu
School of Computer Science &Technology, South China University of Technology, Guangzhou 510006, China

Xifeng Huang, Wenhao Chen
School of Software Engineering, South China University of Technology, Guangzhou 510006, China

***Abstract*—Query processing has attracted significant attention since XML database is proposed. This paper focuses on structural join mainly in our native XML storage and query system (XSQS). A hybrid structural join strategy based on tree-merge algorithm and stack-tree algorithm is proposed. Moreover, a path index scheme is designed to accelerate the query processing. Experimental results show that the proposed hybrid strategy and the index have a good performance in query processing of XSQS.**

***Index Terms*—Query Processing, Structural Join, Path Index, XSQS**

I. Introduction

XML [1] has become the standard in data representation and data exchange on web. XML data grows exponentially every year, making the native XML database attracting more and more attention. Large numbers of research papers have been published on native XML database, particularly in the aspect of query processing, which is one of the most active research topics.

Query processing strategies generally include navigation-based strategy, join-based strategy and the mixed mode strategy. XSQS (XML Storage and Query System) applies the join-based processing strategy, in particular structural join. The main idea of structural join is decomposing a complex query pattern into several sets of basic binary structure relationships, evaluating the binary structure relationships, and then combining the matched results. In this processing strategy, evaluating the basic structures of the relationships is the key issue, which includes parent-child relationships and the ancestor-descendant relationships. In order to improve the efficiency of query processing, developing structural join algorithm is a crucial issue, because a proper algorithm that fits the query processor can reduce the I/O or the time complexity. The contributions of this paper are summarized as follows:

(1) The native XML database XSQS is introduced, with general description of the query processing.
(2) We realize a hybrid structural join strategy on XSQS. Accordingly, a modified tree-merge algorithm for matching parent-child relationships, and a modified stack-tree algorithm for ancestor-descendant relationships are proposed. Moreover, a path index based on hierarchy encoding scheme is developed, which aims at an efficient access to the elements satisfying specific conditions.
(3) Finally, experiments are conducted to show the performance of the proposed approaches in XSQS.

The rest of the paper is organized as follows. Section 2, presents related work, followed by the general overview on query processing emphasizing on the proposed index model in XSQS in section 3. Section 4, describes the structural join algorithms. In section 5, experiments are conducted to evaluate the proposed query processing strategy. Finally, a summary concludes this paper in the last section.

II. Related Work

Structural join is the key issue in join-based query processing strategy, and a lot of papers have been proposed to develop an effective structural join algorithm so far. S. Al-Khalifa et al. [2] proposed a Tree-Merge algorithm, and C. Zhang et al. [3] developed an algorithm named Multi-Predicate Merge Join. Both of them used some conditions to guide the join process, targeting at reducing repeating scan of the candidate node lists. Besides, S. Al-Khalifa et al. [2] proposed another algorithm called Stack-Tree which was based on Tree-Merge algorithm. The Stack-Tree algorithm keeps a stack to store ancestor nodes, in order to get only one scan of the ancestor node list and the descendant node list. S. Y. Chien et al. [4] improved Stack-Tree algorithm by applying B+ tree indexing, through which the invalid nodes can be skipped. All of the above structural join algorithms judge the relationships by using an encoding

doi:10.4304/jsw.8.1.200-207

schema. One of the most common encoding schemes is region-based encoding schema, which is similar to our encoding scheme in XSQS. There are some novel labeling schemes that support the update operations, like CFE [5] or VLEI [6]. They avoid re-labeling nodes to efficiently process the updates, which is also our future work needs to do.

Applying efficient structural join algorithm is one way to improve query processing, and another approach is developing a summary index for query processor. Summary index establishes a simplified XML tree based on paths occurred in XML documents, and as a result, there are not existing two nodes that share the same path in the tree. DataGuide is probably the first summary index, which is proposed in [7], every label path in it occurs once and only once. B. F. Cooper et al. [8, 9] presented Index Fabric index from Patricia Trie tree, by marking every label path with a string encoding, and then inserting the string encoding into Patricia Trie tree. When the query processor executes queries, it turns out to do the string matching. For the purpose of adapting to dynamic changes of queries, making the high frequency queries have better performance, C. W. Chung et al. [10] came up with APEX index. APEX introduces a hash structure to store label nodes in high frequency queries, it is similar to the effect of Caching. When evaluating a new query, the query processor will search hash table firstly to find whether there are target nodes.

III. Query Processing In XSQS

Before going into the query processing in XSQS, a general overview of new version XSQS is shown in Figure 1(see [11] for details of previous version), only the modules corresponding to the query processing are described. The architecture of XSQS mainly consists of several parts: XML loader, storage manager, query engine and index manager. Considering the intimate connection between XML loader and storage manager, we will introduce them together in the following part.

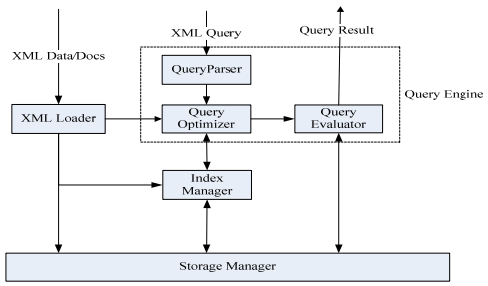


Figure 1. XSQS's Architecture

A. The XML Loader and Storage Manager

To store the XML data, XSQS designs a model of pages managing based on the compressed table of information. It stores two parts of information: the control information of XML documents and XML document data in files.

The XML loader (see [12] for details) is designed to efficiently parse XML data. It uses the SAX parser to analyze XML data and the corresponding information is passed to our DOM tree-like structure (an extension of the W3C DOM interface) to rebuild a DOM tree. During the period of the processing, a thread management is applied to improve the performance of loading.

When finishing the DOM tree, the storage manager traverses the tree to generate three parts of information: *symbols data*, *string data* and *node records*. *Symbols data* specifies the label names occur in XML document, while *string data* saves the actual data value of document and *node records* record all nodes of document. *Node records* keep the document structure information which is the key to reconstruct a XML tree. XSQS denotes nodes by a region-based encode scheme <sIndex, eIndex, level> (the first region-based encode scheme is proposed in [13]), which stores the location of node in XML tree. The node's sIndex represents the value of pre-order traversal (e.g., sIndex of root node is 1), the eIndex represents the maximum sIndex of this node's subtree and the level is the depth of node in XML tree (e.g., level starts from 1). Therefore, it's convenient to judge the relationships between two nodes A and D:

ancestor-descendant relationships: A.sIndex < D.sIndex and A.eIndex > D.eIndex and A.level < D.level

parent-child relationships: A.sIndex < D.sIndex and A.eIndex > D.eIndex and A.level +1= D.level

B. The Query Engine

The query engine manages the query processing in XSQS, the general procedure includes query parsing and query execution (query optimization is an important part of query engine, but it is outside the scope of this paper, so we exclude it), as shown in Figure 2. Firstly, the query engine calls parser interface to analyze the query statement into an abstract query tree. After that, it calls the Tree Walker interface recursively to execute the processing taking the abstract tree as input. Finally, the Result Set interface is called to generate result sets.

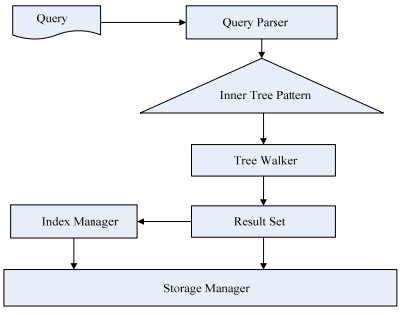


Figure 2. Query Engine

In XSQS, JavaCC (Java Compiler Compiler) is applied to parse the queries, which is a popular open-source grammar generator. If the query is "/site/regions//item [quantity=1]/name", then an abstract tree is generated after calling the parser, is shown as Figure 3.

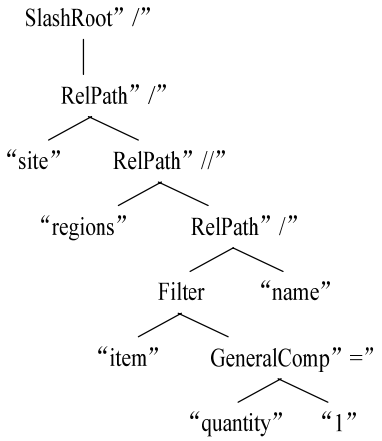


Figure 3. Abstract Query Tree

This query execution is implemented by Tree Walker, whose function is traversing the query tree and calling Result Set to get results. It's a left-deep traversal of the abstract tree, but a right association operation to get query results. When traversing the abstract tree, Tree Walker uses its own query result set to generate some new query result sets, called intermediate results, and then it filters the intermediate results recursively until getting the final results.

Result Set is the interface to get data from storage manager (or the index manager) and provide join operation functions. It keeps the results in a structure named IntArray which can be seen as a two-triple list. Each element stores two values of a node, which are the node's sIndex and its parent node's sIndex, denoted as <node.sIndex, parentnode.sIndex>. Considering the recursive processing of getting the final results, IntArray is more properly represented as <node.sIndex, ancestornode.sIndex>. For example, the query "/site/regions//item [quantity=1]/name" will get final result IntArray as <node ("name").sIndex, node (parent of "site").sIndex> (here node (parent of "site").sIndex=0). IntArray is the main structure for structural join operations. To reduce the accesses to disk, we use the structural join algorithm, and a detail description is given in section 4.

XSQS considers seven types of join operations in query plan, which are summarized in Table I, including the corresponding functions defined in Result Set.

C. Index Manager

When evaluating the query, Result Set can get the data directly from storage manager or through the index manager. Calling the storage manager to get nodes matched a given name is a traversal processing through accessing all nodes stored in the data file to find the matched ones. However, index manager builds a path index tree to provide a way to obtain matched nodes for the given name, moreover, the set of nodes are conforming to the query statement. In that case, index manager has greatly reduced the size of intermediate results, making the number of join operations much smaller.

The main idea of path index is making use of path information in XML document tree, combining the nodes that share the same path as a new one. In that condition, each node of the path index tree definitely has a path and the only. A hierarchy encoding scheme is used in path index tree to represent the path information of node. Unlike the region-based encode scheme (<sIndex, eIndex, level>), hierarchy encoding is a kind of width-first encoding scheme. The definition is as follows:

(1) If N is the root node of a XML document tree, its hierarchy encoding ID is 0, Hid (N) =0;

(2) If N is a child of root node, its hierarchy encoding ID is a binary sequence according to its position among all its siblings. That is, if N is the i^{th} (starting from 0) distinct element in this level of the XML document tree, then N's hierarchy encoding ID is a binary sequence whose i^{th} bit ,from the right side to left side, of the binary sequence is set to 1, while all other bits are set zeros.

(3) If N is a node not in condition (1) and (2), its hierarchy encoding ID is a binary sequence made up of two parts, S1 and S2 (Hid (N) =S1·S2), in which S1 is the encoding ID of N's parent and S2 is the binary sequence representation for i which is the i^{th} child among all the siblings in the same level.

Through the hierarchy encoding scheme, it is convenient to judge the relationships between XML nodes (for details see [11, 12]).

TABLE I.

JOIN OPERATIONS IN XSQS

No.	Join Type	function in Result Set
1	A / B	newNamedParentofNamedChild
2	A / ResultSet	namedParentofEvaledChild
3	ResultSet / B	namedChildofEvaledParent
4	ResultSet / ResultSet	evalParent
5	A // ResultSet	namedAncestor
6	ResultSet // ResultSet	evalAncestor
7	ResultSet[B] or ResultSet [@B]	newLeafNodeList

After the XML loader parsers XML document into DOM tree, index manager generates path index tree according to the DOM tree, with a hierarchy encode for each tree node. During the structural join process, list of sIndexs that represent specific tree nodes are served as input to Result Set. So, if Result Set wants to get the nodes by index manager, there is a necessary process that path tree need to do. That is, corresponding the hierarchy encode to a list of sIndexs and keeping the list in path tree. This process can be seen in Figure 4.

Index manager provides supper set of accurate results for query. Taking the query "/site/regions//item [quantity=1]/name" for example. Assuming the situation that Result Set needs to get nodes from data file, whose name equals "name". If Result Set gets nodes from storage manager, then it will get all the nodes that have the name "name", while index manager will get nodes that have the name "name" and under the path "/site/regions//item [quantity]" (not path "/site/regions//item [quantity=1]/name", because the path index doesn't support predication operation), and there is an extra filtering operation " item [quantity=1]" needs be done. Supposing another query "/site/regions//item/name", then index manager can directly get the final result set, that is, the set of element nodes named "name" and under the path"/site/regions//item".

```
1. Algorithm: MatchsIndex ( PathTree, sIndexList)
2. Inputs: PathTree, sIndexList: represented as <sIndex, tagName, parentsIndex>;
3. Outputs: The path tree with the matched sindex list in each node;
4. Procedure:
    for(each node Si in sIndexList)
        Pi = findPathTreeNode(PathTree, Si.parentsIndex)
        for(each node Pci in Pi.childList)
            if(Pci.tagName = Si.tagName)
                add Si.sIndex to Pci.sIndexList

1. Algorithm: findPathTreeNode(PathTree,Si.parentsIndex)
2. Inputs: PathTree, Si.parentsIndex;
3. Outputs: Pi : the corresponding node in PathTree;
4. Procedure:
    max_size = maxsIndex+1
    //If node N's sIndex is n, Hid(N)=h, then sIndextoEid [n]=h;
    sIndextoEid[max_size]
    for(each node Pi in PathTree)
        if(Pi.eid = sIndextoEid [Si.parent])
            return Pi
```

Figure 4.Algorithm: Matching Hierarchy encode with sIndex

IV. STRUCTURAL JOIN

As mentioned above, IntArray is the main structure for structural join in XSQS, and each element in it is the form of <node.sIndex, ancestornode.sIndex>. Furthermore, IntArray is ordered by sIndex when it is generated from data file.

Considering IntArray DList (d1, d2,..., dm) in size m and IntArray AList (a1,a2,…,an) in size n, represent two lists of candidate nodes as inputs for structural join, then the output is a new IntArray resultList that satisfies parent-child (or ancestor-descendant) relationships. The resultList is sorted by child (descendant) node's sIndex in ascending order.

The traditional way for structural join is a traversal-style algorithm. That is, for each node a_i in AList, judging relationships between a_i and all nodes in DList. In that case, the parent-child relationships join has the time complexity O(m*n). As for the ancestor-descendant relationships, structural join needs to get all ancestor nodes of the descendant node, by recursively accessing data file through its parent node's sIndex, and then checks the given node to see whether it is one of them. This process makes ancestor-descendant relationships join have time complexity O(m*n*h) and the I/O complexity O(m*h), h represents average depth of node in the XML tree. The time complexity of structural join is too high to accept, therefore some algorithms are considered to reduce it.

A. Modified Tree-Merge-Desc Algorithm

Tree-Merge-Desc algorithm [2] is the first try to achieve low time complexity. After a deep analyze on the algorithm, some modifications are done to adapt the query processing in XSQS. The main idea of Tree-Merge algorithm is finding the first ancestor node a_i in AList that may satisfy the relationships for each node d_j in DList, which is named scan start point. As for the node d_{j+1}, it scans from scan start point of d_j to get its own scan start point. There exists a condition in XSQS that d_{j+1}.sIndex >= d_j.sIndex but d_{j+1}.parent-sIndex<= d_j.parent-sIndex (it means d_{j+1}.parent node is listed before the start scan point), then the Tree-Merge algorithm will skip the d_{j+1}'s parent node. So, we do some modifications to avoid this situation. Figure 5 shows the modified algorithm for parent-child relationships joins in XSQS and Figure 6 shows the modified algorithm for ancestor-descendant relationships joins in XSQS.

The modified Tree-Merge-Desc algorithm in XSQS has the time complexity O(m+n) for parent-child relationships join. However, for the ancestor-descendant relationships joins, the time complexity is O(m*n*h) and the I/O complexity is O(m*h), which are still unacceptable. We will discuss the improvement later.

```
1. Algorithm: Modified Tree-Merge-Desc
2. Inputs: AList, DList
3. Outputs: resultList
4. Procedure:
   first-anc= AList.firstNode
   for(d= DList.firstNode; d!=null; d= d.nextNode){
      // An adding condition
      while (first-anc>0&& d.parent-sIndex< first-anc.sIndex) first-anc - -;
      //find the first node that can be parent of d
      for (a= first-anc; a!=null && d.parent-sIndex >a.sIndex; a= a.nextNode);
      first-anc=a;
      if (d.parent-sIndex =a.sIndex)
         Append (d.sIndex, a.parent-sIndex) to resultList;
   }
```

Figure 5.Modified Tree-Merge-Desc for Parent/Child

```
1. Algorithm: Modified Tree-Merge-Desc
2. Inputs: AList, DList
3. Outputs: resultList
4. Procedure:
 first-anc= AList.firstNode
 for(d= DList.firstNode; d!=null; d= d.nextNode){
      Anclist[ ]=get_ancestorlist(d);   //get all ancestors of node d
      //find the first node that can't be ancestor of d
      for(a= first-anc; a!=null && d.parent-sIndex>=a.sIndex; a= a.nextNode);
      first-anc=a;
      for(Anclist[i]) {
        for(a=AList.firstNode ;a<first-anc; a=a. nextNode){
          [if (a.sIndex= Anclist[i]) ]
          Append( d.sIndex, a.parent-sIndex) to resultList; }
      }
}
```

Figure 6. Modified Tree-Merge-Desc for Ancestor/Descendant

B. Modified Stack-Tree- Desc Algorithm

Judging ancestor-descendant relationships by Tree-Merge-Desc algorithm still has a high complexity, which only uses parent node's sIndex. Therefore, another algorithm named Stack-Tree-Desc [2] is developed with the encode scheme <sIndex, eIndex, level> to judge ancestor-descendant relationships. The traditional Stack-Tree algorithm specifies only one condition in structural join (AList and DList are not empty or the stack is not empty), this consideration is so rough that may make mistake. Therefore, we consider some special situations may occur in the algorithm, adding the proper procedure for each one to deal with. See the details of the modified algorithm in Figure 7.

The I/O complexity of the algorithm is O(m) and time complexity is O(m+n), which are acceptable.

V. EXPERIMENTAL EVALUATION

In this section, we present experiments to verify the validity of the approaches in XSQS. All experiments were run on a machine with 2.53GHz Intel(R) Core(TM) i3 processor, 8GB memory, and the OS is win7.

The popular XMark [14] data set are used in these experiments, with the size from 10MB to 40MB by 10MB increments. We evaluate the structural join algorithms in XSQS with or without path index, to evaluate the performance of algorithms in different environment, furthermore, to figure out how the algorithms and index contribute to query performance. The experiments use sets of queries, shown in Table II. Q1 to Q3 have two ancestor-descendant relationships join operations in similar length. Q4 to Q6 have ancestor-descendant relationships join operations from one to three in similar length. This classification is used to present whether the algorithm has similar performance in a variety of queries.

Here, the MST is short for Modified Stack-Tree algorithm, MTM for Modified Tree-Merge algorithm, and PI for Path Index.

Figure 8 depicts the speedup ration of MST over MTM for Q1 to Q6 with PI (or without PI) in database size of 20MB as an example. The experiments in database of other sizes show the similar effects. It is concluded that no matter PI is used in XSQS or not, MST always has a much better performance than MTM.

To figure out the correlation between databases' size and speedup ration of MST over MTM, we did the tests with six queries in different database size. The speedup ratio is computed as the value of *(the cost time of MTM – the cost time of MST)/ the cost time of MTM*. Figure 9 shows that the speedup ratio generally increases along with the growth of the size of database.

```
1.  Algorithm: Modified Stack-Tree-Desc
2.  Inputs: AList, DList
3.  Outputs: resultList
4.  Procedure:
    a= AList.firstNode; d=DList.firstNode
    while (AList and DList are not empty or the stack is not empty){
  // Consider several situations may occur, adding the proper procedure for each one
     if (AList and DList are not empty){
        if (stack is empty){ stack. push(a); a=a.nextNode;
        } else {
           If (d.sIndex<stack->top.sIndex) d=d.nextNode;
           else{
              if (a.sIndex> stack->top.eIndex &&
                  (d.sIndex< stack->top.sIndex || d.sIndex< stack->top.sIndex) ) stack. pop ();
              else if( a.sIndex<d.sIndex){ stack. push(a); a=a.nextNode;}
                   else{
                      for( a_i in stack) Append( d.sIndex, a_i.parent-sIndex) to resultList; }
           }
        }
      }else if (AList is empty and DList is not empty){
         While( DList and stack are not empty){
           If( d.sIndex>= stack->top.sIndex && d.eIndex<= stack->top.eIndex)
             { for( a_i in stack) Append( d.sIndex, a_i.parent-sIndex) to resultLis;}
           else stack. pop(); }
     }else break;
    }
```

Figure 7. Modified Stack-Tree -Desc for Ancestor/Descendant

In Figure 10 and Figure 11, “MTM (with/without) PI” represents the speedup ratio of MTM with PI over MTM without PI, and the same explanation applied to “MST (with/without) PI”. “(MST/MTM) without PI” represents the speedup ratio of MST without PI over MTM without PI, and the same explanation applied to “(MST/MTM) with PI”.

From Figure 10 and Figure 11, we can see the correlation between PI and MST (or MTM). It is concluded that if PI has greater influence on MST than MTM (the speedup ratio of MST with PI over MST without PI, is bigger than the speedup ratio of MTM with PI over MTM without PI), then the speedup ratio of MST over MTM will increase when using PI. To the opposite, if PI has greater influence on MTM than MST (the speedup ratio of MTM with PI over MTM without PI, is bigger than the speedup ratio of MST with PI over MST without PI), then the speedup ratio of MST over MTM will decrease when using PI. Figure 10 shows the situation one, and Figure 11 shows the other situation.

TABLE II.

QUERIES FOR TESTING

QNum	Path Expression
Q1	/site/regions//item/description//parlist/listitem/text/emph
Q2	/site/regions//item/description/parlist/listitem//parlist/listitem
Q3	/site//annotation//parlist/listitem/parlist/listitem
Q4	/site/closed_auctions/closed_auction/annotation/description//parlist/listitem
Q5	/site/closed_auctions/closed_auction//description//parlist/listitem

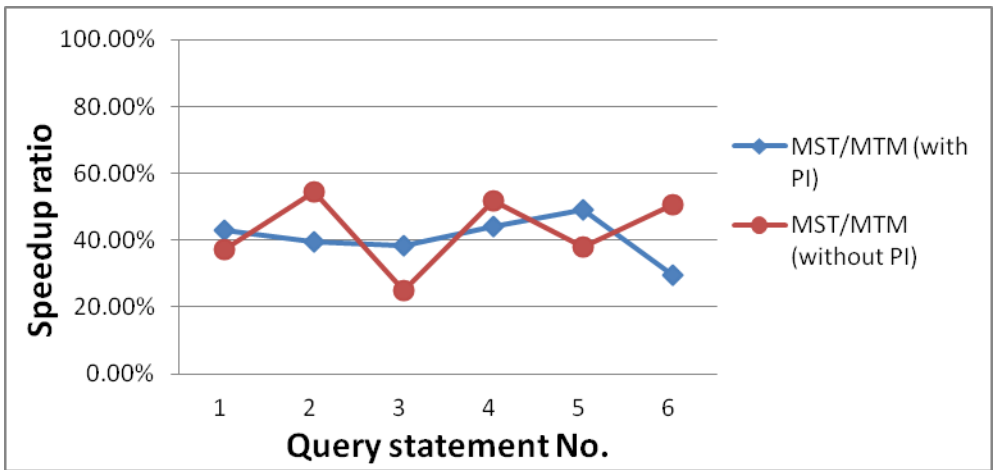


Figure 8. Speedup Ratios of MST over MTM for Q1-Q6 (with/without) PI

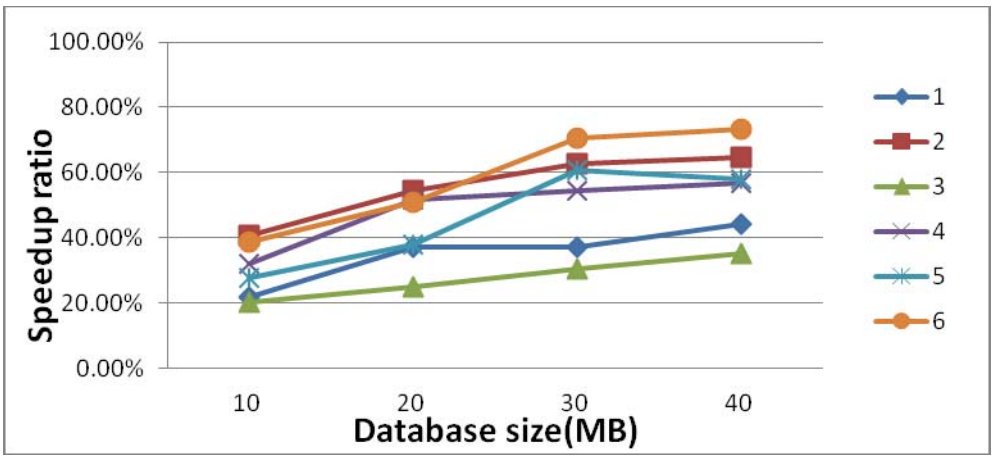


Figure 9. Speedup Ratios of MST over MTM for Q1-Q6（without PI）

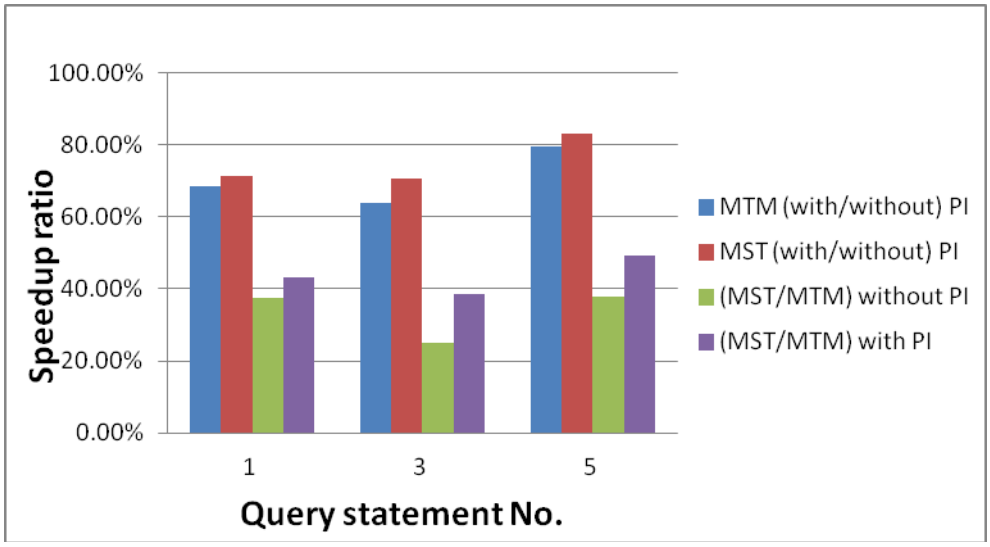


Figure 10. Speedup Ratios of Q1, Q3, and Q5 with Different Algorithms in Database Size of 20MB

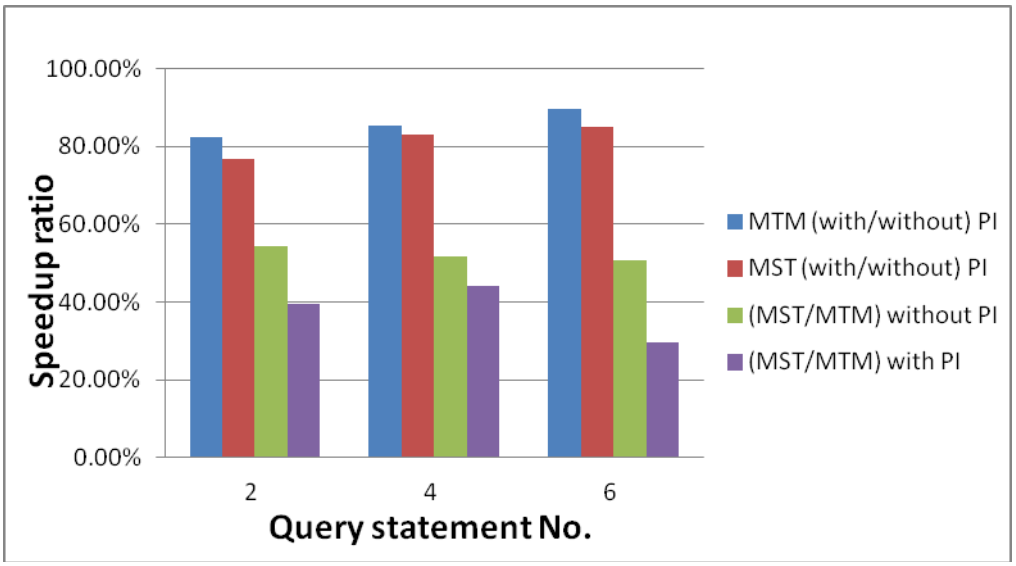


Figure 11. Speedup Ratios of Q2, Q4, and Q6 with Different Algorithms in Database Size of 20MB

VI. Conclusions And Future Work

In this paper, we introduce the native XML database system XSQS and focus on implementation of the query processing in it. Moreover, we show the details on how the structural join and the path index is designed to help query engine get a better performance. The experimental results indicate that the Modified Stack-Tree algorithm always has a much better performance than Modified Tree-Merge algorithm, and the advantages generally increase along with the growth of the size of database. Our future work will focus on the query optimization in XSQS by taking advantages of structural join.

Acknowledgment

The work is partially supported by National Natural Science Foundation of China (Grant No. 71090403), Education Ministry of Education of P.R.C (Grand No. x2rjB7110020), Bureau of Science and Information Technology of Guangzhou (Grant No. x2rjB2111420).

References

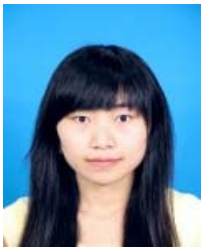
[1] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation, 26 November 2008, DOI= http://www.w3.org/TR/REC-xml/.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching", In Proc. 18th ICDE Conf, Mar 2002, pp. 141-154.

[3] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems", In Proc. of the 2001 ACM SIGMOD Int'l Conf. on Management of Data, May 2001, ACM Press, pp. 425-436.
[4] S. Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents", In Proc. of the 28th Int'l Conf. on Very Large Data Bases, August 2002, pp. 263-274.
[5] Yi Jiang, Xiangjian He, Fan Lin, et al., " An Encoding and Labeling Scheme Based on Continued Fraction for Dynamic XML", Journal of Software, Vol 6, No 10 (2011), Oct 2011, pp. 2043-2049
[6] Jie Chen, Wenxin Liang, Haruo Yokota, " A Two-Dimension XML Encoding Method based on Variable Length Binary Code", Journal of Software, Vol 6, No 12 (2011), Dec 2011, pp. 2426-2433
[7] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases", In Proc. of the 23th VLDB Conf, August 1997, pp. 436-445.
[8] B. F. Cooper, N. Sample, M. J. Franklin, et al., "A Fast Index for Semistructured Data", In Proc. of the 27th VLDB Conf, September 2001, pp. 341-350.
[9] B. F. Cooper, N. Sample and M. Shadmon, "A Parallel Index for Semistructured Data", In Proc. of the 2002 ACM Symposium on Applied Computing (SAC), March 2002, ACM Press, pp. 890-896.
[10] C. W. Chung, J. K. Min and K. Shim, "APEX: An Adaptive Path Index for XML Data", In Proc. of 2002 ACM International Conference on Management of Data SIGMOD, June 2002, ACM Press, pp. 121-132.
[11] L. Hong, D. Li, N. Gu, "Design and Implementation of XSQS System", In Proc. of the 2009 International Forum on Information Technology and Applications, Vol. 3, 2009, pp. 385-388.
[12] D. Li, N. Gu, "Hierarchy Encoding Based XML Query Estimation", In International Forum on Information Technology and Applications, Vol. 2, May 2009, pp. 451-456.
[13] P. F. Dietz, "Maintaining Order In a Linked List", In Proc. of the 14th annual ACM Symposium on Theory of Computing, 1982, ACM Press, pp. 122-127.
[14] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey and R. Busse, "XMark: A Benchmark for XML Data Management", In Proc. of International Conference on Very Large Data Bases, 2002, pp. 974-985.

Dong Li Dong Li received his BS from Harbin Institute of Technology in 1992 and his Master and PhD from Huazhong University of Science and Technology in 1995 and 2001, respectively. He had been to UCSB as a visiting scholar from 2008 to 2009. He is a professor at South China University of Technology in China. His research interests include XML databases, mobile database and service computing.

Xiuyu Lu Xiuyu Lu received her BS from Sun Yat-sen University in 2010. She is currently a Master student in South China University of Technology in China, and her research interest is XML database.



Xifeng Huang Xifeng Huang is currently a Master student in South China University of Technology in China, and her research interest is XML database.

Wenhao Chen Wenhao Chen received his BS from South China University of Technology in 2011. He is currently a Master student in South China University of Technology in China, and his research interest is XML database.