XML Data Storage and Query Optimization in Relational Database by XPath Processing Model

Xiaojie Yuan
College of Information Technology and Science, Nankai University, Tianjin, China
Email: yuanxj@nankai.edu.cn

Xiangyu Hu, Dongxing Wu, Haiwei Zhang[+] and Xin Lian
College of Information Technology and Science, Nankai University, Tianjin, China
Email: {huxiangyu, wudongxing, zhanghaiwei, lianxin }@dbis.nankai.edu.cn

***Abstract*—XML is de facto new standard for data representation and exchanging on the web. Along with the growth of XML data, traditional relational databases support XML data processing across-the-board. Consistent storage and efficient query for XML data is the chief problem in XML supported relational databases. This work presents mechanisms of Storage and query optimization for XML data in relational database. XML data are treated as a kind of data type in relational database, and XML tables are used to store native XML data in fixed schema. Structural summary index is built and maintained in relational database and an optimizing mechanism based on XPath model named Compressed XML Query Tree will also be presented in order to improve efficiency of XML data query by reducing superabundant join operations from ancestor-descendent axis. All strategies are appropriate for classical XML query algorithms. Algorithms for XML query will be performed in experiments on real XML datasets in relational database and query workloads to report the performance of our mechanism and show the efficiency compared with other mechanisms.**

***Index Terms*—XML, XPath, Data Storage, Query Optimization, Compressed XPath Query Tree**

I. INTRODUCTION

With the rapid growing popularity of XML to present data, XML has become a standard format to store data in many areas and share data between them. Though XML has been used in many domains for data exchanging and representation, a great deal of XML data appears and how to manage XML data efficiently has becoming a hotspot for researchers.

Native XML databases are powerful system for XML data management. Thus databases can only process pure XML data and cannot management other common data, such as relational data. On the other hand, relational Databases are traditional solutions for data management because of their mature theoretic system and manufacture. As a result, almost all relational databases, such as SQLServer, Oracle, DB2, have begun to support XML data in recent years. All of them treat XML data as a new data type and store in special column of relations and can perform XQuery and XPath[1][2] for XML data query.

Processing XML data has extended ability of relational databases for various data management.

XQuery and XPath are most important XML query languages. XPath is included in XQuery and usually appears in XQuery expressions while querying XML data. In native XML databases, XML query languages can be directly used to obtain results because there is integrated hierarchy of grammar analysis and query execution in such system. In relation databases, XML query languages cannot be performed, they must be integrated with SQL. SQL/XML standard provides rules for the integration of XQuery and SQL. According to SQL/XML, new grammar of data query will be added to SQL and XML query will be achieved in relational databases.

Data Storage and query optimizing are important problems in research of database technology. During the research of XML data management, how to realize XML data Storage and query optimization in relational database is worthy of study.

A mechanism of XML data Storage fitting for relational database will be suggested in this paper and an efficient XPath query optimization mechanism based on data Storage that makes full use of XML indices to quickly retrieve XML data. Coming together with the mechanism, the Compressed XPath Query Tree (CXQT) based on indices is proposed, which significantly reduces many join operations brought by PC relations in XPath expressions. Then query algorithm based on CXQT is presented to deal with all the structural relationship using our mechanism.

II. RELATED WORK

Intuitionistic model of XML data is DOM tree. All XML components including elements, attributes and text has been converted to nodes in DOM tree. Navigating XML data in tree model by XPath is common solution of query. Unfortunately, DOM tree cannot stored in relational databases directly. As a result, XML data must be stored in relational databases by other solutions.

[+] corresponding author: Haiwei Zhang

doi:10.4304/jsw.8.4.809-816

Almost all popular relational databases have begun to support XML data by adding a new data type named XML, such as SQLServer, Oracle and DB2. SQLServer treated XML data as Binary Large Object (BLOB), and indices on XML data can be built by ODRPATH[3] coding. Oracle used object-relation model to integrate XML data management. XML data has been mapping to relations, and XML schema also has been mapping to relational schema, object-relation model could be built consequently. XML data completely has been processed in relational mechanism. DB2 used different solution to manage XML data. DB2 integrated native mode for XML data management. Relational data and XML data used different engine to process, therefore, DB2 has not really compromised these two kinds of data.

The key issue in XPath query processing is the matching of structural relationship. There are two kinds of structural relations in XPath expressions, parent-child (PC) and ancestor-descendant (AD). C. Zhang et al.[4], combining with the traditional thinking of merging algorithm and interval encoding, proposed Multi-Predicate Merge Join (MPMGJN) algorithm which is the first solution to dual structural join. MPMGJN algorithm reduces the comparison between XML nodes by their document order relationship. However, S. Al-Khalifa et al.[5] found MPMGJN algorithm would spend a lot of time in some cases when issuing the structural relationship, and proposed StackTree algorithm which used a stack to save the nodes with the same tag and have the AD relationship in one path of XML document tree. Consequently, the comparison between nodes will happen on the top of stack. Since then, the application of stack plays an important role in XPath query algorithm. Nevertheless, MPMGJN algorithm and StackTree algorithm may produce very large intermediate results. To solve this problem, N. Bruno et al.[6] proposed holistic twig join algorithm that maps XPath query tree into linked-stacks, and then by recursion checking structures in streams pushes the nodes matching the structural relationship into stacks, finally gets the result in a leaf-to-root path. However, if there are a large number of nodes with PC relationship in the twig queries, the TwigStack algorithm not only have to create a large number of stacks but also need to match the PC relationship. Besides, most of existing XPath query algorithm suppose the XML data is outside the DBMS, so those algorithms would have a very large cost on space and time to retrieve XML data. This retrieving method has become the bottleneck of structural join. Based on algorithm above, optimization mechanisms for improving query efficiency have been presented in recent years[7][8][9][10], these methods focused on native XML data query without any solutions derived from relational databases. X. Yuan et al.[11] proposed structural index, which has been used in XPath query algorithm. However it didn't solve the problems in query rewriting based on structural index.

III. XML DATA STORAGE AND INDICES

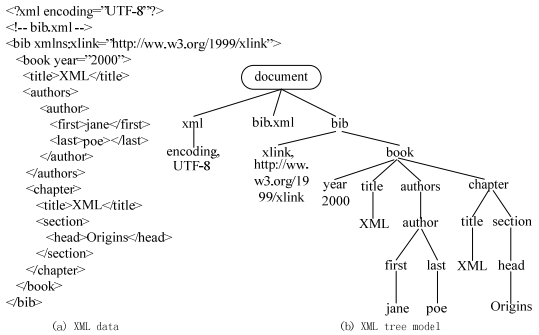


Figure.1. XML data and tree model

XML can be represented by an ordered label tree as figure 1 shows. Tree model is widely used for XML representing. Indices for XML data are usually built on tree model or other models transformed from tree model. In relational database, XML data usually is stored in tuples as attribute with a specific data type. As presented above, BLOB, relation mapping are used in popular relation databases. This paper proposes XML tables for data Storage in fixed schema. XML tables are treated as system tables and not accessed by users directly with SQL. Data type named xml is also used in this mechanism and relates to XML tables for XML data management.

A. XML Table

Tables are particular data structure in relational database. In order to make full use of capability for data management, integrating XML data Storage into relational database and managing these two categories of data together can exploit their advantages to the full. XML tables are specific data structure for XML data Storage. XML tables will be stored in relational database and used to stored pure XML data.

Definition 1. XML Table is used to store XML data as relational pattern with fixed schema.

Figure.1 is an example of XML data named bib.xml and its tree model. Table 1 shows the XML table for data in bib.xml. The schema of XML tables is:

(nodeid, type, name, value, path)

where nodeid is primary key.

- **DLN Code**

Column nodeid stores DLN code[12] of each XML node in tuples. DLN code is a kind of prefix code encoding XML nodes in documental order. It encodes parent node as the prefix of its child nodes. And the first node in XML data is encoded by number 1. The code of a node is smaller than its right sibling nodes and bigger than its left sibling nodes. As a result, DLN codes can describe relation between XML nodes distinctly, and can be used in all kinds of computation for structural relations of XML nodes.

- **Node Type**

Column type describes type of XML node stored in the XML table. There are three kinds of XML node: element node, attribute node and text node. Each type relates to a kind of component of XML data.

TABLE I.
XML TABLE OF BIB.XML

NodeID	Type	Name	Val	Path
1	D			
1.1	E	1		1
1.1.1	E	2		2.1
1.1.1.1	A	3		3.2.1
1.1.1.2	E	4		4.2.1
1.1.1.2.1	T		XML	
1.1.1.3	E	5		5.2.1
1.1.1.4	E	4		4.2.1
…				

- **Reversed Path**

Column path describes path of the XML node in the XML table. The path stored in this column is an order reversed path. For example, if path of an XML node is book/authors/author, the value in the cell is author.authors.book. Numbers can be designed for labels of XML nodes, so path of an XML node can be represented by a sequence of number as Table I shows.

XML table stores pure XML data as tuples, each node of XML data is treated as a tuple. Along with the increment of nodes in XML table, Using XML table, efficiently accessing specific node in large quantities of nodes is difficult. As a result, indices of XML table are necessary for the mechanism.

B. XML Indices

In order to improve efficiency for XML query, indices are built and maintained in relational databases. Two kinds of indices are presented in this paper, Basic Label Index (BLI) and Structural Summary Index (SSI). BLI is used for XML data Storage and SSI is used for XML query.

- **Basic Label Index**

Basic Label Index (BLI) is a simple index for XML data, and it can be used while locating XML nodes in XML tables. BLI will map element nodes and attribute nodes to NodeID in XML table by the label of XML nodes. Figure.2 shows the process of mapping from node labels to column NodeID. BLI can be used for indexing labels of element nodes and attribute nodes. In Fig. 2, if given label of attribute nodes named year, collection of nodes encoded by {1.1.1.1} will be located, and values of nodes in the collection will be obtain in the column value of XML table.

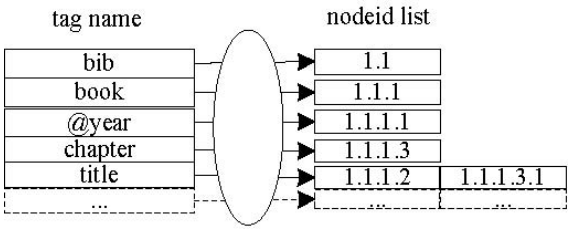


Figure.2. Basic Label Index

- **Structural Summary Index**

BLI can improve efficiency of XML data Storage by mapping labels of nodes. For XML data query, another index named Structural Summary Index (SSI) will be used to improve efficiency of XML data query. SSI is related to paths of XML nodes, as shown in Figure.3. In XML table, column path can provide information for building SSI. Order of nodes in paths of SSI is reversed from path column because of searching efficiency while parsing XML data. Each path in SSI is named Path Index, and locates XML node rapidly. All paths compose of SSI of XML data, and this structure is appropriate for XML query.

SSI indexes XML data by path of nodes. As a result, query XML data by path can get high-efficiency performance, especially using XPath. XPath language is an important and useful XML query language, which was issued by W3C containing in XQuery (XQuery is considered to the most integrated XML query language). In XPath expressions, ancestor-descendent (AD) and parent-child (PC) axes are important, they describe PC relations and AD relations between XML nodes respectively. Parent-child axis is related to paths in SSI, using SSI can get more effective query pressing. For another important axis--ancestor-descendent axis in XPath, some popular query algorithms such as TwigStack, performed efficiently on such structural relations. In order to make full use of high-efficient algorithms, a particular model named Compressed XML Query Tree will be presented in this paper.

IV. XPATH EVALUATION AND OPTIMIZATION

Using XML tables, XML data can be stored as relational modes. XML data query can be performed when relational database could parse XQuery language. In order to improve efficiency of XML data query in relational database, mechanism for optimizing XPath evaluation in XQuery expressions will be presented in this section.

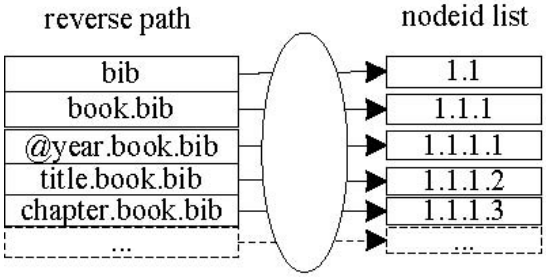


Figure.3. Structural Summary Index

A. XPath Analysis

XPath is frequently-used XML query language. XPath is composed of expression named XPath expression. Location path is the most important expression of XPath. Location paths compose of steps, each step is related to a node along with axis or predicates. When querying XML data by XPath, expressions will be firstly parsed to XPath Grammar Tree (XGT). For example, XPath expression:

/book[//title = "XML"]/authors/author[first = "john"]

will be parsed to XGT as shown in Figure. 4. Then XGT will be stored in memory as XPath Query Tree (XQT). Each step of XPath expression will be a node named XQ_AxisStep in XQT and the node includes three pointers: nodeTest, axisType and predicateList. All pointers relate to components of XPath expression, such as axis, predicates and node test. XGT shown in Figure.4 can be stored as XQT shown in Figure.5.

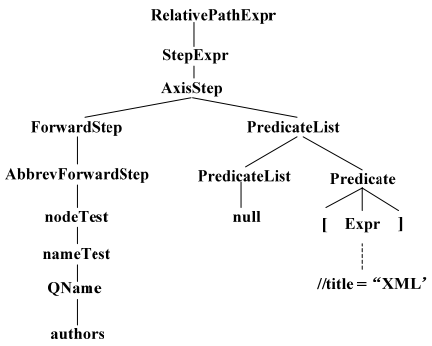


Figure.4. XPath Grammar Tree

B. Compressed XPath Query Tree

XPath expression can be represented by XQT for XML data query. But XQT has its' own disadvantages in XML query. Firstly, XQT has many levels and is difficult to access nodes in the lower levels. Secondly, the structure of XQT is not fit for SSI and cannot make full use of indices in XML query. Lastly, XQT preserves PC relations in XPath expression, but it can cause difficulty of processing AD relations and algorithms such as TwigStack cannot be performed well.

In order to improve adaptability of query algorithms, Compressed XPath Query Tree (CXQT) model will be used for XPath.

Definition 2. Compressed XPath Query Tree (CXQT) is transformed from XQT by compressing PC relations in XPath expressions. CXQT changes PC relations into AD relations, and then it can fit for more query algorithms than XQT.

There are three kinds of nodes in CXQT, BLI nodes, SSI nodes and output nodes. All of these nodes can make control of query executing and save results of XML query.

- BLI nodes are obtained directly by BLI and labeled by the same description of original nodes of XML data.
- SSI nodes are obtained by SSI. Label of SSI nodes is a sequence of XML nodes and usually denotes a part of XPath expression as location path. Relations between SSI nodes are usually parent-child.
- Output nodes can be BLI nodes or SSI nodes. They point out query nodes which must output query results in CXQT, so they are also named result nodes. When output nodes are SSI nodes, the last step in location path of SSI is considered as output nodes of CXQT.

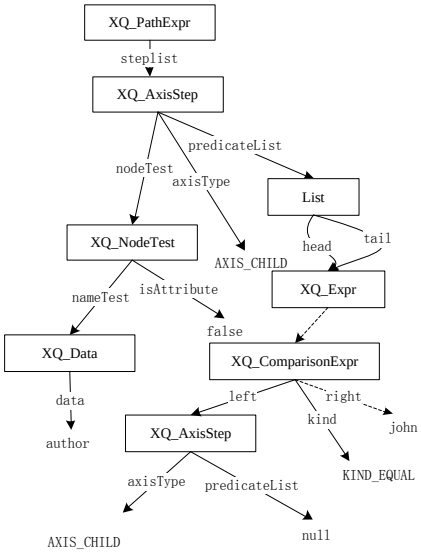


Figure.5. XPath Query Tree

There is a relationship named Generation-Gap in CXQTs.

Definition 3. Generation-Gap Relation exists among some nodes in CXQTs that the results of querying these nodes have certain generation gaps.

For example, PC relation is one of Generation-Gap relations that its generation gap is one. However, AD relation is not a Generation-Gap relation since it doesn't have a certain generation gap.

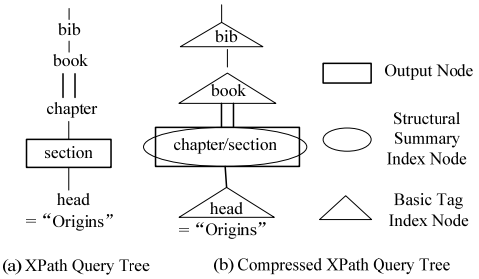


Figure.6. Rewriting SXPE for CXQT

CXQT model can be used to evaluate two kinds of XPath expressions, simple XPath expression and twig XPath expression.

- **Simple XPath Evaluation**

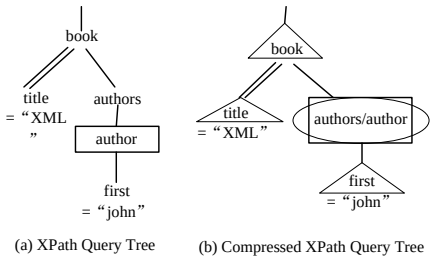
Simple XPath Expressions (SXPEs) are XPath expressions without twigs. SXPE composes of one or more step expressions, and structural relations in every two adjacent steps are PC or AD. For example, XPath expression:

bib/book//chapter/section[head ="Origins"]

can be represented as shown in Fig. 6, in which PC relation is represented by single line and AD relation is represented by double lines. Output nodes in the figure are represented by rectangle.

If SXPE has only one element node or attribute node, result nodes will be obtained by BLI, and the node will be rewritten to be BLI node. While more than one element nodes appear in SXPE and structural relations of these nodes are PC, the nodes will be merged to a path. SSI will be used to obtain output nodes for merged path.

Occasionally, SXPEs contain output nodes or non-end nodes with predicates. These nodes will be used as separator to divide SXPE into two parts and these two parts of location path will engender Generation-Gap. While SXPEs include AD relations, the location path will also be divided into two paths and these two paths will be considered to be rewritten.



(a) XPath Query Tree (b) Compressed XPath Query Tree

Figure.7. Rewriting SXPE for CXQT

SXPE are divided into two parts according to such nodes: output nodes, nodes with predicates and nodes with AD relation. And these two parts of SXPE will recursively invoke function rewriteSimplePath to obtain local rewritten results. Finally, the results will be merged and returned. When SXPE has only one node, the node will be changed to a BLI node. When SXPE has more than one node which cannot be divided, then the nodes will be merged to a SSI node.

In Figure 6, nodes bib, book and head are BLI nodes and node chapter/section is SSI node and output node at the same. The CXQT model changed from XQT of Figure. 6(a) is shown in Figure. 6(b). CXQT can use indices reasonably and reduce times of structure join.

- **Twig XPath Evaluation**

Twig XPath Expressions (TXPE) are XPath expressions with twig, they usually need to choose nodes satisfying to some tree structure. For example, following TXPE:

/book[//title = "XML"]/authors/author[first = "john"]

will query such node:

(i) Value of node first is john, and first is sub child of author

(ii) Element node author is child node of authors, and parent of authors is book. And the element node book has a descendent element node labeled title whose value is XML.

Dividing TXPEs will consider nodes with predicates in location path. And these nodes are computed as context node-set of next step or predicates. Nodes with predicates cannot be merged while their location is non-endpoints. For example, element book in Fig. 7(a) includes predicate //title="XML", it cannot be merged to SSI node labeled /book/authors/author, but can be considered as a BLI node. Paths belong to node book are all SXPEs. Using algorithm for SXPE rewriting, CXQT will generate as Figure. 7(b) shows.

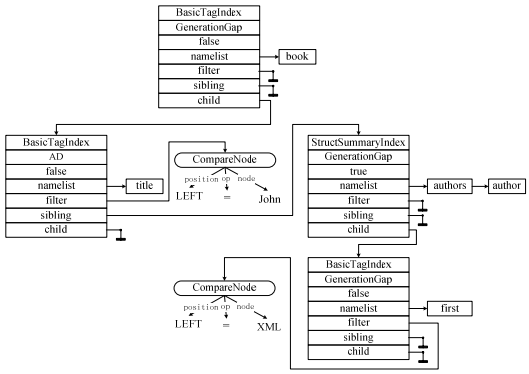


Figure.8. Data structure of CXQT.

Algorithm for TXPEs rewriting will divide TXPEs into two parts, one is a SXPE and the other are more than one TXPEs. Using function rewriteSimplePath for each TXPE can obtain rewriting results, and then merge to CXQT as a tree node. Data structure of CXQT is shown in Figure 8. CXQT uses binary tree to represent multi-way tree by linked list. Labels will be considered as nodes of linked list namelist and structural pointer will be stored in domain filter.

C. Query Achievement

CXQT will be used to query XML with the help of DLN code. There are two kinds of XML query strategies, join-based query strategy and holistic-compare-based query strategy.

Structural relations exist between adjacent steps of XPath expression. Each step is related to a collection of XML nodes. Dividing XPath expression into segments, all these segments have structural relations. Each segment can be considered as a minimum query unit, and can get query results by query algorithms. All results obtained by segments of XPaths will be merged by structural relations and then final results of XML query will generate. The strategy above is join-based query.

CXQT based on SSI can divide XPath expression into several segments. XPath segments can obtain nodes set by BLI and SSI. Typical algorithms for execution XML query by join are named structure join, such as multi-predicate merge join (MPMGJN) and StackTree, are fit for join-based XML query.

TXPE will be divided into many dual structural join when using join based XML query. Then intermediate results will be joined in order to get final results. While join list is in big size, intermediate results will occupy a great deal of memory space and bring out high cost of I/O between memory and disk. To solve this problem, Bruno et.al presented holistic twig, the method can process all dual structural join in one path of query tree once off. And the strategy is holistic-compare-based query. XML query by holistic twig can be performed by algorithm named PathStack and TwigStack.

V. Experimental Evaluation

We performed our experiments on an Intel Core 2 Duo 1.86GHz with 2GB of RAM, running MS Windows 7 and Fedora 12. XML data was stored in relational database PostgreSQL and XPath query processing was also implemented in PostgreSQL. Every query was executed three times, and the last two measurements were averaged to the reported execution time. Consequently, the warm cache was used in query execution. With a cold cache all results were greatly different from the latter ones and are not presented due to lack of space.

A. Datasets

The experimental evaluation used a set of synthetic and real datasets[13] which contain a wide range of XML's characteristics (Table 2). We used ToXgene to generate ten XML documents as the synthetic dataset and adopted the DBLP dataset as the real dataset.

TABLE II.
XML Data Sets

DataSet	Size(M)	Nodes	Max/Avg Depth
DBLP	127	6.34M	6/2.9
ToXgene(1)	2	0.16 M	5/5
ToXgene(2)	4	0.31 M	
ToXgene(3)	6	0.47 M	
ToXgene(4)	8	0.62 M	
ToXgene(5)	10	0.78 M	
ToXgene(6)	20	1.56 M	
ToXgene(7)	40	3.13 M	
ToXgene(8)	60	4.69 M	
ToXgene(9)	80	6.26 M	
ToXgene(10)	100	7.82 M	

The synthetic dataset has a smooth structure that each node sets with different labels have the same size. However, the real dataset is made up with many node sets distinct in label name and size.

Table 3 shows all the XPath queries used for the experiments. We selected only one XPath query to run on the ToXgene dataset to evaluate the efficiency of structural index, and five queries to run on the DBLP dataset to evaluate the algorithms based on CXQT. These queries, containing four simple XPath queries and two twig queries, have different combinations of PC and AD axes and different selectivity over the datasets.

B. Evaluating the Efficiency of Structural Index

We first compared MPMGJN Algorithm without structural index with CXQT-based MPMGJN Algorithm for processing the same simple XPath query, Q1, in Table 3 over the same dataset, ToXgene dataset, in Table 2. For each execution of query in different XML documents of ToXgene dataset, we recorded the query processing time for the two algorithms.

TABLE III.
Queries used for the Experimental Evaluation

	DataSet	Query
Q1	ToXgene	//book/author//item1
Q2	DBLP	/dblp/book/title
Q3	DBLP	//article//sub/i
Q4	DBLP	/dblp//article[//cite]/title
Q5	DBLP	//inproceedings[year]//title/sup
Q6	DBLP	//inproceedings[year][.//title/sub]/pages

Figure. 9 depicts the performance results based on the query Q1 on ToXgene dataset. The MPMGJN Algorithm without structural index spends more time on all executions than CXQT-based MPMGJN Algorithm. The cause is (i) it is faster to retrieve required nodes with structural index, (ii) the CXQT reduces the XQT, so Q1 is consist of two query nodes based on CXQT but three nodes based on XQT. Besides, the executing times of these two algorithms all have a rapid increment when querying over large XML documents. It dues to large data causes the gigantic I/O costs.

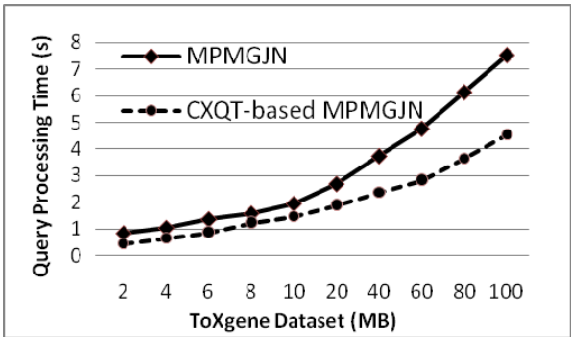


Figure.9. Evaluating the Efficiency of Structural Index

C. Evaluating the CXQT-based XPath Processing Algorithm

In this section, we will study the performance of the XPath processing algorithms based on CXQT.

As the processing of twig XPath queries can adopt different processing algorithms compared with the processing of simple XPath queries. The experiments were separately conducted on simple XPath queries and twig XPath queries. In final part of this section, we make a comparison of our XPath query mechanism and the

XPath query mechanism in SQL Server 2008, since these two mechanisms have the similar storage and index methods.

- **Experiments on Simple XPath Queries**

Figure 10 depicts the results on processing simple XPath queries with different CXQT-based algorithms. When executing Q1 and Q2 which have only one query node and small query result set, CXQT-based MPMGJN algorithm and CXQT-based StackTree algorithm return the query result set immediately using structural index. Nonetheless, Q3 contains three query nodes which have a large XML node set, consequently CXQT-based MPMGJN not only has to do a lot of comparison but also spends numerous costs on disk I/O.

When there is only one query node in CXQT, the join-based query strategy is prior to the holistic-compare-based query strategy, since the result is returned immediately with structural index in join-based query strategy, while the holistic-compare-based query strategy needs to make the result go through the stack bringing more cost in time. When the number of query nodes in CXQT is two, the efficiencies of the three algorithms are almost equal because they all get the result in one time scan of the candidate lists. However, when CXQT carries more than two query nodes, holistic-compare-based strategy is obviously more efficient than join-based strategy which spends more than five times the time.

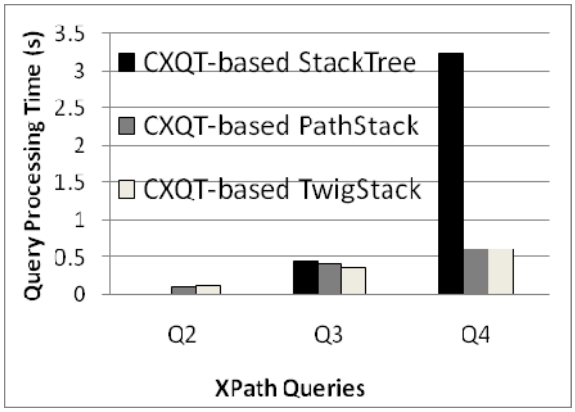


Figure.10. Simple XPath Queries.

- **Experiments on Twig XPath Queries**

Figure.11 depicts the results on processing twig XPath queries with CXQT-based StackTree algorithm and CXQT-based TwigStack algorithm. For the query processing time, the CXQT-based TwigStack algorithm significantly outperforms the CXQT-based StackTree algorithm, and is three to five times faster than the CXQT-based StackTree algorithm. Since TwigStack algorithm greatly declines the time by matching CXQT in streams, resulting in a linear cost in time. Meanwhile, TwigStack saves the I/O cost in large join operations.

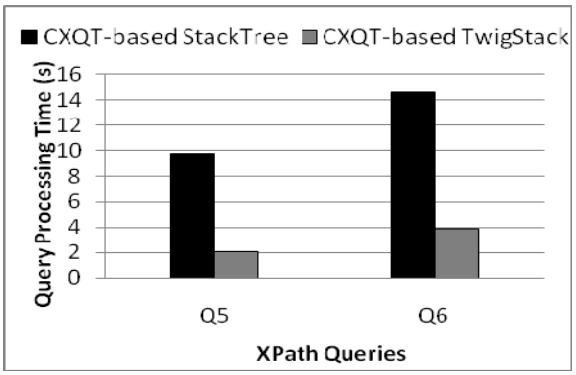


Figure.11. Twig XPath Queries.

- **Comparison with SQL Server 2008**

XML data can be stored and queried in SQL Server 2008. The storage of XML data in SQL Server 2008 has some kind of similarity with the storage system in this paper which stores XML data in relational database and keeps the structural characteristics of XML data. We make the comparison with SQL Server 2008 to prove that our XPath processing mechanism has some advantages in comparison with that in SQL Server 2008.

We stored the DBLP dataset in Table 2 into SQL Server 2008, and then executed the Q2-6 in Table 4. In order to prevent the cost in printing results, we used count functions, so that the time will be recorded is the query processing time of XPath.

TABLE IV.
COMPARISON WITH SQL SERVER 2008

	CXQT-based TwigStack	SQL Server 2008
Q2	112ms	4s
Q3	368ms	9s
Q4	605ms	54m39s
Q5	2097ms	1m42s
Q6	3913ms	18m44s

VI. CONCLUSIONS

This paper proposes an efficient mechanism of XML data Storage and query optimization in relational databases. XML data is stored as XML tables, which fit for relational databases. Basic Label Index and Structural Summary Index can be built based on XML tables and using Compressed XPath Query Tree can make efficient performance on XML data query by reducing many join operations. Algorithms for XML data query can be well performed using solutions we presented. Experiment results show mechanism presented in this paper will get better performance than other mechanisms.

ACKNOWLEDGMENT

This work is supported by National High-tech Research and Development Program (863 Program) of

China (Program No.:2009AA01Z152) and the NSFC (Grant Nos. 60973089).

REFERENCES

[1] Mary Fernandez, Ashok Malhotra, et al. XQuery 1.0 and XPath 2.0 Data Model. (2007). Available from http://www.w3.org/TR/xpath-datamodel/.

[2] 2. Denise Draper, Peter Fankhauser, et al. XQuery 1.0 and XPath 2.0 Formal Semantics. (2007). Available from http://www.w3.org/TR/xquery-semantics/.

[3] 3. Patrick O'Neil, Elizabeth O'Neil, et al. ORDPATHs: Insert-Friendly XML Node Labels. Proceedings of ACM SIGMOD, (2004) June 13-18; Paris, France.

[4] 4. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. Proceedings of ACM SIGMOD, (2001) May 21-24; Santa Barbara, California, USA.

[5] 5. S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A Primitive for Efficient XML Query Pattern Matching. Proceedings of the 18th International Conference on Data Engineering, (2002) Feb.26-Mar.1; San Jose, California, USA.

[6] 6. N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. Proceedings of ACM SIGMOD, (2002) June 3-6; Madison, Wisconsin, USA.

[7] 7. J. Lu, Tok W. Ling, C. Y. Chan, T. Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. Proceedings of VLDB, (2005) Aug.30-Sep.2; Trondheim, Norway.

[8] 8. Zhifeng Bao, Tok Wang Ling, et al. SemanticTwig: A Semantic Approach to Optimize XML Query Processing. Proceedings of Database Systems for Advanced Applications (DASFAA), (2008) Mar. 19-21; New Delhi, India.

[9] 9. H. Georgiadis, M. Charalambides, V. Vassalos: Cost Based Plan Selection for XPath. Proceedings of ACM SIGMOD, (2009) June 29-July 2; Providence, USA.

[10] 10. H. Georgiadis, M. Charalambides, V. Vassalos: Efficient Physical Operators for Cost-based XPath Execution. Proceedings of 13th International Conference on Extending Database Technology (EDBT), (2010) March 22-26; Lausanne, Switzerland.

[11] 11. X. Yuan, X. Wang, and C. Wang. Efficient XPath Evaluation Using a Structural Summary Index. Proceedings of International Conference on Computer Science and Software Engineering (CSSE), (2008) Dec.12-14; Wuhan, Hubei, China.

[12] 12. Bohme T,Rahm E. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. Proceedings of 3rd International Workshop Data Integration over the Web (DIWeb), (2004) June 8; Riga, Latvia.

[13] 13. University of Washington. XML Data Repository. www.cs.washington.edu/research/xmldatasets.