International Journal of Database Theory and Application
Vol.9, No.10 (2016), pp.99-110
http://dx.doi.org/10.14257/ijdta.2016.9.10.09

Query XML Streaming Data with List

He Zhixue[1, 2] and Liao Husheng[1]

[1]*Department of Computer Science, Beijing University of Technology, Beijing, China*
[2]*Computer and Remote Sensing Information Technology Institute, North China Institute of Aerospace Engineering, Langfang, China*
hezhixue@gmail.com

Abstract

There has been a growing practical need for querying XML streaming data efficiently. Stream requires to be read sequentially and only once into memory, the query must be processed on the fly. QXSList technique is proposed for massive data processing, which takes the SAX events sequence as input, buffer the incoming elements for further processing, remove unnecessary elements from the buffer in time, and give the results on the fly. Data model and algorithm integrated framework are defined, the integrate methods of how to process predicate and wildcard are discussed respectively. Level value is used for determining the relationship of two elements and relational pointers are constructed for linking multi lists in this method. The experimental results show that our approach is effective and efficient on this problem, and outperforms the state-of-the-art algorithms and query engines especially for data size is very large. At the same time, memory usage is nearly constant.

Keywords*: streaming data, query processing, XPath query, relational pointer*

1. Introduction

With the development of mobile internet, internet of things, and many intelligent terminal applications, many stream data have been produced by data monitoring in environment and traffic flow, real-time analysis in financial stock market trend and public interest, *etc.* Streaming data are different from traditional structure data for they arrive continuously, data process require only once sequential scan, and return results on the fly. XML has become the de facto standard for data representation and exchanging on web due to its flexibility of organizing data such as self-describing capability, flexible organization. The purpose of querying data streams is to identify specific data patterns in the continuous flow of data which match queries user presented. The problem of XPath [1] query evaluation against streamed XML data is a fundamental database problem in the context of XML.

In context of XPath streaming evaluation, there are two problems are commonly studied [2]. One is stream filtering problem that for given a set of queries, the filter will determine which of them have a nonempty output on an incoming XML document stream, filtering systems such as YFilter [3] and SFilter [4]. The other is the stream querying problem which finds all the matches of the output nodes of a given set of queries against the XML stream. Among querying XML stream algorithms, Peng *et al.* developed an automaton-based XPath stream-querying system XSQ [5], it needs to explicitly enumerate and store all pattern matches for an input query during its execution. TurboXPath [6] is a representative of the array-based approach, it first builds a parse tree for a given query and then finds matches of the parse tree nodes on the input stream, however it only works efficiently for queries on non-recursive XML documents and exhibits exponential behavior for queries on recursive documents. Some stack-based algorithms extends the

ISSN: 2005-4270 IJDTA

array-based algorithms by exploiting the path stack technique to compactly encode a potentially exponential number of query pattern matches in polynomial space, which include TwigM [7], StreamTX [8] and LQ and EQ [9]. XQStream++ [10] addressed the problems of processing tuple extraction queries for streaming XML, but it is not suitable for deep recursive document, predicate and wildcard, this problem is analyzed in Section 2.3.

The main contribution of this paper is summarized below:

(1) We propose an algorithm named as QXSList, which takes the SAX events sequence as input, buffer the incoming elements for further processing, remove unnecessary elements from the buffer in time, and give the results on the fly.

(2) We give the specific implementation of this algorithm to process two XPath query fragments: $XP^{\{/, //, []\}}$ and $XP^{\{/, //, *\}}$ respectively.

(3) Our extensive performance evaluation shows that our algorithm has considerable practical performance advantages over state-of-the-art stream-querying algorithms and engines.

The rest of the paper is organized as follows. We first describe some background information in Section 2. After that we present the framework and implementation of the algorithm for processing XML streaming data with list and relational pointer, in Section 3. An experimental study is provided in Section 4, and concludes our work in Section 5.

2. Preliminaries and Problem Analysis

In this section, we introduce the basic terminology and notation. We first define the event-based XML streaming evaluation model and the XPath query fragment. Then, we analyze the problem existing in methods based on stack process streaming and propose our solution.

2.1. Data Model

An XML document is commonly modeled as a rooted, labeled and ordered tree. Each node in the tree corresponds to an element or a value, and the edges represent (direct) element-subelement or element-value relationships. Figure 1(b) shows the XML tree corresponding to the XML document of Figure 1(a), subscript is added to differentiate same label node. This data model requires entire document to be read into memory to construct DOM tree, but in big data era, many files become very large, the size scale even reach terabytes(TB). It exceeds the storage capacity of memory and at the same time, there are many data which are not relevant to user's query, so it is not feasible to load all data in processing. Streaming form XML data are parsed in order, typically as a sequence of SAX [11] event following preorder traversal of DOM tree. The XML element in Figure 1(a) arrive in the order of a1, b1, b2, b3, d1, d2, e1, e2, c1, and c2. If we define SE(e) is the start element event of e, and EE(e) is the end element event of e, then, the corresponding sequence of SAX events of XML document in Figure 1(a) is SE(a1), SE(b1), SE(b2), SE(b3), SE(d1), SE(d2), SE(e1), EE(e1), EE(d2), SE(e2), EE(e2), EE(d1), EE(b3), SE(c1), EE(c1), EE(b2), SE(c2), EE(c2), EE(b1), and EE(a1).

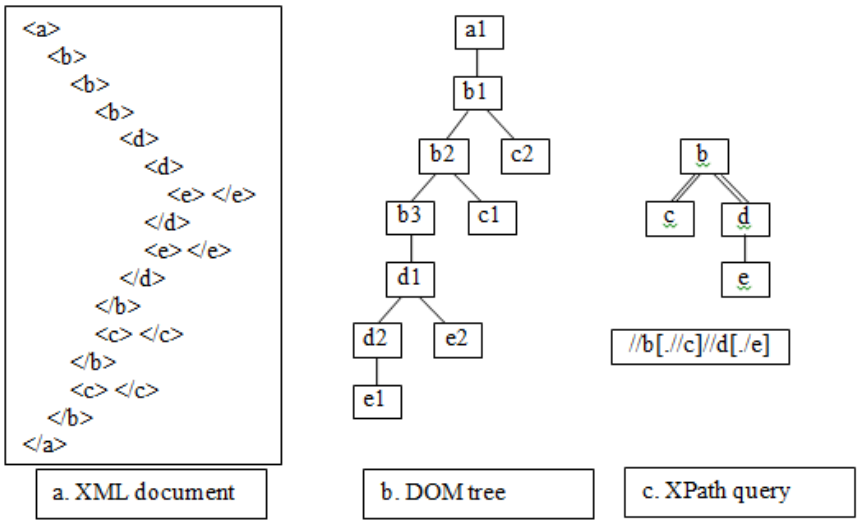


Figure 1. XML Data Mode and XPath Query

2.2. XPath Query

XPath is a standards query languages for XML, since it has been used in many XML applications and in some other languages for querying and transforming XML data, such as XQuery and XSLT. In this paper, we propose practical methods to deal with different fragments of XPath. An XPath query is expressed as twig patterns, an edge can be either single-lined "/" or double-lined "//", which constraints the two matched nodes is either a PC (Parent-Child) relationship or an AD (Ancestor-Descendant) relationship. Figure 1(c) shows the twig pattern of XPath query Q1: //b[.//c]//d[./e]. Further, we define four attribute w.r.t a given query node e in query Q, e.cp as e's closest parent, e.ca as e's closest ancestor, e.cfr as e's closest forward relationship, and e.cbr as e's all closest backward relationships. For example, in Q1, b.cbr are: (1)a descendant c as a predicate; (2)a descendant d as a output node. Following, we focus on a practical fragment of XPath defined as follows:

Path := Step | Path Step
Step := Axis NodeTest | Axis NodeTest '['Predicate']'
Axis := '/' | '//'
NodeTest := name | *
Predicate := Path | Predicate 'and' Predicate | Predicate 'or' Predicate | 'not' Predicate

2.3. Problem Analysis

In streaming data process, predicate ([]) and wildcard (*) need to be considered comprehensive. For instance, Q1: //b[.//c]//d[./e] in Figure 1(c), which include "//" axis (A-D relationship), '/' axis (P-C relationship), and "[]"predicate, matches the XML document in Figure 1(a), after event SE(a1), SE(b1), SE(b2), SE(b3), element b1, b2 and b3 are candidates for query node b. If stored in stack as in XQStream++ [10], SE(d1) makes d1 as a candidate for query node d, it is a descendant of b1, b2 and b3, but only top element b3 in stack can be visited, the relationships between d1 and b1/b2 cannot maintained. Similar situation for P-C relationship between d and e query nodes, as shown in Figure 2, d2 and e1 are satisfy P-C relationship, on receiving SE(e2), d1 and e2 are also satisfy P-C relationship, but now d1 is at the bottom of stack, if operations sequence of stack are strict, d1 cannot be visited again until d2 has been pop from stack. This phenomenon is common in recursive XML documents. Recursion, where some elements with the same name are nested on the same path in the data tree, occurs frequently in

XML data in practice. For instance, among 60 DTDs surveyed in [12], 35 are recursive. In this paper we propose an algorithm to process streaming data with list and relational pointer based on SAX parsing, which is verified efficient and effective for XPath query fragments.

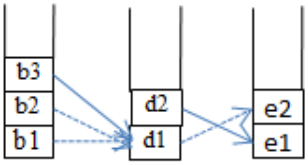


Figure 2. Buffer Elements Using Stack

3. Qxslist Algorithm Design and Implementation

In this section, we first give the framework of QXSList algorithm and then present detailed implementation for processing XPath fragments $XP^{\{/,\ //,\ [\ ]\}}$ and $XP^{\{/,\ //,\ *\}}$. At the last, we propose some suggestions of optimized methods to further improve efficiency and reduce buffer size.

3.1. Framework of Algorithm

Processing XML stream data queries with list and relational pointer is driven by event which produced by SAX parse XML. Different events trigger corresponding operations, and the algorithm is implemented as shown is Figure 3. QXSList takes XML stream and XPath query as input parameter, process the query until it reads the end of the stream, and outputs matched results as soon as they are identified during query processing. At the start element event in SAX, we can get the element information includes label and level value, the relationship between elements can be calculated by the level.

Poperty1: Two elements x and y satisfy A-D relationship if and only if y is included by x, that is y is in the scope of x's start and end of label; if the y's level value is lager one than x, then they are P-C relationship.

For example, in Figure 1(b), a1's level value is 1, and d1's level is 5, at the same time d1 is included in the start and end label of a1, so d1 is a descendant of a1; similarly, b1's level is 2, and difference between a1 and b1 is 1, so they are P-C relationship.

QXSList uses list as buffer data structure, each item in list is a tri-tuple made up by element information: <label, level, pointers>, where the "pointers" is an array records all the relationship pointers. The algorithm implementation is shown in Figure 3, it creates a list for every query node of user posed query (line 01), constructs a work stack for judging the element relationships (line 02) and initializes a variable level to record element layer value (line 03). At the SE event of an element, level value is added by 1, call createPointer function to construct relationships between current visited element and elements that have been stored in list (line 08). When all list is not empty, it gives the match results on-the-fly (line09-10). At the EE event, level is subtracted by one, and checkList function removes the elements which will not be used in follow processing to reduce the buffer size. For XPath query fragment $XP^{\{/,\ //,\ []\}}$ and $XP^{\{/,\ //,\ []\}}$, the procedures are same, but createPointer and checkList should be considered for how to deal with predicate([]) and wildcard (*) respectively.


```
Algorithm QXSList
Input: XML data stream D, and query Q
Output: matched output elements
01 create ListN for every query node n;
02 create a work stack WS;
03 level = 0;
04 while (not visit end of the stream)
05    e = visiting element;
06    if (at the start event of e)
07       level = level + 1;
08       createPointer(e);
09       if (all list are not empty)
10          output the target nodes;
11    if (at the end event of e)
12       level = level -1:
```

Figure 3. Framework of QXSList Algorithm

3.2. Xpath Fragment $XP^{\{/, //, []\}}$ Query Processing

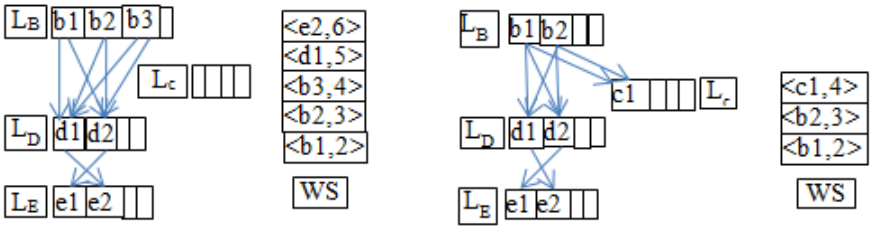
Process $XP^{\{/, //, []\}}$ query focuses on different situation for predicate. For example, in query Q1: //b[.//c]//d[./e], node b should satisfy(1)has a descendant node labeled by c and (2)has a descendant node labeled by d, but for the order of c and d is non-determined, c maybe before d or after d, so all the possibility should be considered. Function createPointer constructs the relationships between elements in different list through relation pointers, and the implementation is shown in Figure 4.

```
createPointer(x)
01 if (x is not a candidate for a query node)
02    return;
03 get relationship x.cfr in Q;
04 get WS top element y;
05 if (relationship type is P-C)
06    if (x.level - y.levle == 1)
07       create P-C pointer between y in ListY and
x
08       add x to ListX and push it to WS;
09 else if (relationship type is A-D)
10    if(x.levle – y.level >= 1)
11       get x.ca as z;
12       for each (element z in ListZ)
```

Figure 4. Implementation of Function Createpointer for $XP^{\{/, //, []\}}$

It first filters elements which are not candidates for query nodes (line 01-02). To add relational pointers for the current visited element, the function gets the closest forward relationships of corresponding query node in query (line 03) and the top element in work stack WS. Then two cases are processed respectively. One is for P-C relationship, it needs to calculate the level value difference is one, if meets this requirement, then a pointer is created between two elements, and the visiting element is added to list and pushed into stack (line 05-08). The other is for A-D relationship, it needs to create a pointer between the visiting element and all its ancestors, the rest operations are similar to first case (line 09-14).

Exaple1: Give the concrete process step as follow: for XPath query Q1: //b[.//c]//d[./e] and XML streaming data in Fig 1(a), when SAX parsing document receive SE(e2), list L_B, L_C, L_D and L_E states are shown in Figure 5(a), element b1, b2 and b3 have been added to L_B, d1 and d2 have been added to L_D, for d1 and d2 are all descendants of b1, b2 and b3, there are multiple pointers are create for their A-D relationship. On receiving SE(e1), it makes element d2's predicate condition changed to be satisfied, at EE(e1) event, e1 is pop from work stack. On receiving SE(e2), d1 and e2 are P-C relationship, and a P-C pointer is added between d1 and e2. Note that at this situation, d1 is the list head, d2 is the list tail, if use stack, then d1 cannot be visited again. Now, there are not element c has been read, L_C is empty. In the procedure, work stack WS store the elements that have been parsed their start labels, but not encountered their end labels.



(a) Lists Status after SE (e2) (b) Lists Status after SE(c1)

Figure 5. Procedure of QXSLis Execution

On receiving EE event of an element, QXSList checks the current visiting and elements that have pointers form it to remove them from list if they are not used in the following process for reducing buffer size. In Figure 6, function checkList firstly filters elements are not candidates (line 01-02), and then pop current element form work stack (line03). Then it gets the back relationship of visiting element in query for checking integrity of its structure that is its predicates and backward structure relationships. If at the end label, this element is not satisfying query specify conditions, then it will have no chance to change its status forever, and can be removed form list (line 05-06). If current element has an integrate structure, the function gets the closest ancestor with predicates of this element corresponding node in query to check the ancestor list elements' predicates condition. If predicates are not satisfied that is having relational pointers linked query node and predicate node, then, current element should be buffered and prepared for visited in following process when predicate arrived to match query. If predicate are already satisfied, it means that the match operation have been executed at the start element event, so the current element and its related children or descendants can be remove form list (07-10). In remove function, it deletes current element and child or descendant elements which it pointed if they are not used by others at the same time.


```
checkList(x)
01 if (x is not a candidate)
02    return;
03 pop x form WS;
04 get relationships x.br in Q;
05 if (some relationships have not corresponding pointer)
06    remove(x);
07 else
08    get query nodes with predicate before x in Q;
09    if (all have pointers corresponding their predicates)
10       remove(x);
remove(x)
01 for each (element which has a pointer from x)
02    if (it has no other pointers except x)
03       remove it from list;
04 remove x from ListX
```

Figure 6. Implementation of Function Checklist for $XP^{\{/, //, [\,]\}}$

Example 2: Continue Example1, SAX parsing Figure 1 XML data, at the event of EE(d2), because b query node's predicate c is not satisfied, d1 and d2 maybe the candidate answers, they should be buffered. When receiving EE(b3), at this moment, b3's predicate "[//c]" has not been satisfied, it means that b3 is not a candidate for query node b and can be removed form list. But d1 and d2 have other pointers form b1 and b2, they cannot be removed with b3 at the same time. At this time point, the status of all lists is shown in Figure 5(b).

If all lists are not empty, it means that all query nodes with their candidate and relationships are all satisfied, so this algorithm directly output target nodes for answer. Compared with other methods that all have a "match" action to get answers, it checks node test, relationship and predicate at the parsing process and constructs relational pointers, so no needs to travel the candidate nodes again.

3.3. XPath Fragment $XP^{\{/, //, *\}}$ Query Processing

In XPath query, "*" is a wildcard can be match any element, we focus on how to deal with fragment $XP^{\{/, //, *\}}$ in this section. In QXSList algorithm, crate a list for every "*" node, if one node satisfies the relationship of "*", it should be added to List*. In Figure 7 we give the implementation of XCreatePointer function which extended by createPointer function in Figure 4 for processing query includes wildcard but not include predicate. In this function, for every visited element, if appears as a candidate for a determined query node, then it is handled by createPointer function (line01-02). Then it also needs to be checked for wildcard node's structure relationship. We also have to consider two situations according to its forward relationship in query: P-C and A-D (line03-14). It is similar to common query node, and the only difference is we do not care its label, but only focus on its level.

```
XCreatePonter(x)
01 if (x is a candidate for a query node)
02    createPointer(x);
03 get forward relationship with * in Q
04 if (relationship is P-C)
05    get *.cp list's tail element as y
06    if (x.level - y.levle == 1)
07       create P-C pointer between y and x;
08       add x to List*;
09 else if (relationship is A-D)
10    for each (*'ca list's element y)
11       if (x.levle – y.level >= 1)
12          create a A-D pointer between y and x;
13    if (exit one or more pointer to x)
14       add x to List*;
```

Figure 7. Implementation of Function Xcreatepointer for XP$^{\{/, //, *\}}$

In this fragment, there is no predicate, so at the end element when remove elements form list, we only need to check if this element is also in List* except is included in query node list. The detail implementation is shown in Figure 8.

```
checklist(x)
01 if (x is a candidate of a query node)
02    remove x from ListX;
03 if (List*'s tail element is x)
04    remove x from List*;
```

Figure 8. Implementation of Function CheckList for XP$^{\{/, //, *\}}$

3.4. Optimal Method

For XML streaming data processing, runtime and buffer size are two important metrics to measure an algorithm's performance, here we propose some optimal ideas for QXSList, and in Section 4, we will give the experimental results and analysis.

Predicate in a query should be evaluated only once, that is only if predicate is satisfied, then other elements matched this predicate can be ignored. To improve the efficiency, query node with predicate can be added a flag variable to record whether predicates are satisfied or not. After this flag is labeled as "true", predicate nodes will not be processed again.

There are two data structures in QXSList, one is list and the other is stack. Work stack is used to store elements have been parsed their start labels to calculate relationship between current visiting element and element in stack, but we found that this judgment can be calculated using list items but not stack elements. To reduce the buffer size, the stack can be removed. Function createPointer change as follow: delete line 04, add line 051 and 091 after line 05 and 09 respectively.

```
051 get x.cp list tail element as y
091 get x.ca list tail element as y
```


In this section, we propose methods how to deal with XPath query fragments $XP^{\{/, //, []\}}$ and $XP^{\{/, //, *\}}$, due to the space limit, the method for $XP^{\{/, //, [], *\}}$ has to be omit, and be induced from previous suggestions.

4. Experimental Results and Analysis

4.1. Experimental Setup

In this section, we compare the performance of our method with that of state-of-the-art XPath stream query algorithm XQStream++ and query engines, MonetDB [13] and SAXON [14]. XQStream++ has shown comprehensive performance advantages over many stream-querying systems and the two engines are most recently released for processing XQuery. We implement QXSList and XQStream++ using Java, and run all the experiments on 2.66GHz Interl Core2 Quad CPU with 4GB memory at Window 7 operating system. We test the performance of four methods on two real datasets, DBLP [15] and Treebank [16], and one synthetic dataset, XMark [17]. DBPL is shallow but wide and Treebank is narrow and deeply recursive. XMark is a well-known benchmark dataset. On each of the three dataset, we test 2 typical queries as shown in Table 1. These queries include / and // axes, *-node and multi predicates.

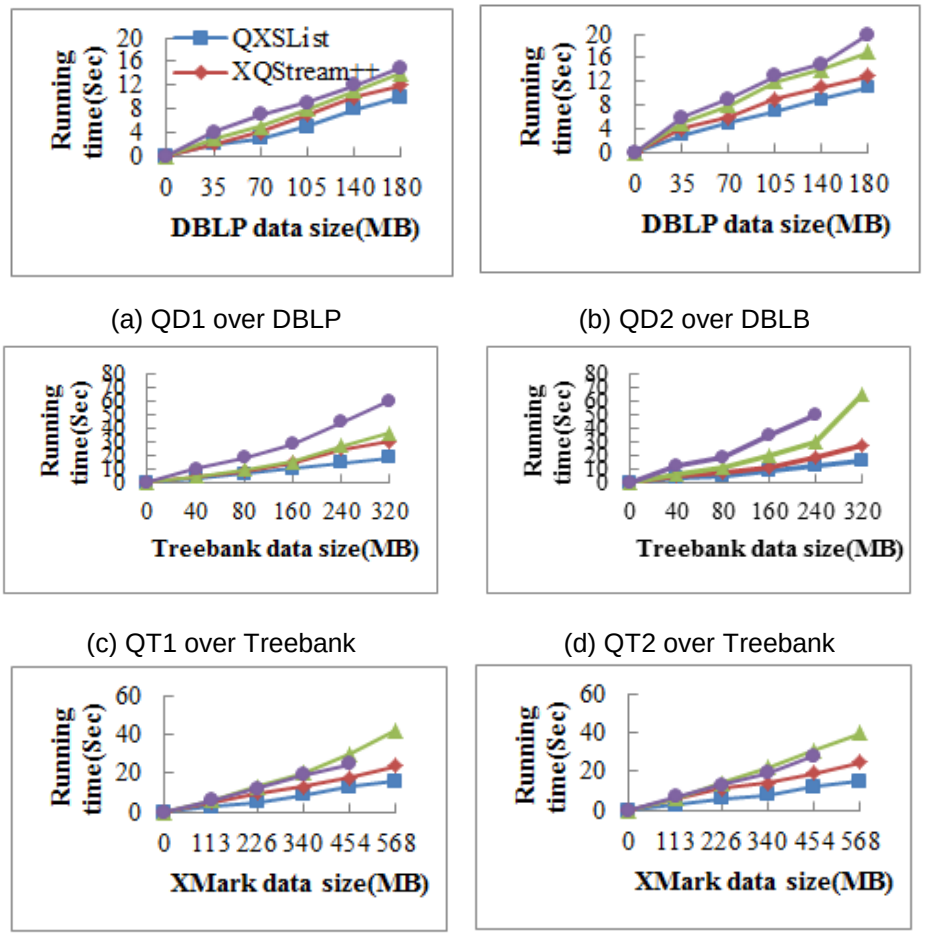
Table1 Test Data Set and Queries

数据集	查询表达式
DBLP	QD1:/dblp/inproceedings[title]/author
	QD2:/dblp/inpriceedings[title and year]/author
TreeBank	QT1://S[.//VP[.//JJ][.//VBD]]//NP[.//WP]//DT
	QT2://S[.//NP[.//DT][.//NN]]//*[.//TO]//NN
XMark	QM1://item[name]//mail[text//emph]
	QM2://closed_auction/annotation/*//emph

We compare these methods in terms of running time and memory usage. Running time is the time spent from the beginning parsing input XML data document to the end element event of last, and memory usage is the maximum usage of main memory that each method consumes during query processing. Each query is executed ten times, and the average value is computed and compared.

4.2. Running Time Performance

Figure 9 shows the results of running time performance comparison over three data sets varying different data size. From this figure, we can see that, QXSList is more efficient than other three methods especially for larger size data document, because when create relational pointers it also match structure relationship, this make it no needs a function to get results but output target elements directly. In Figure 9(a) and (b), the data sizes are relatively small, QXSList's performance is similar to XQStream++, but super than MonetDB and SAXON. With the data size becoming larger, in Figure 9(c-d), and (e-f), the differences become lager, QXSList gains considerable advantages. This comes from that structure match is synchronized with document parsing, and results output on the fly.



(a) QD1 over DBLP (b) QD2 over DBLB

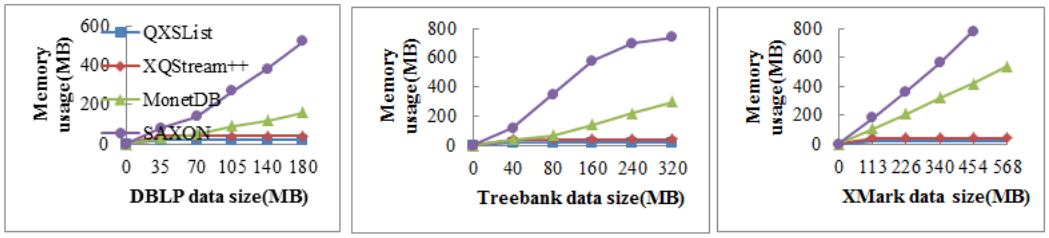
(c) QT1 over Treebank (d) QT2 over Treebank

(e) QM1 over XMark (f) QM2 over XMark

Figure 9. Comparison in Running Time

4.2. Memory Space Performance

Figure 10 shows the performance in terms of memory usage. We plot the results of the first query for each dataset, since other queries are all similar. Among the compared methods, MonetDB and SAXON use a large amount of memory, especially when input data size increased, they may buffer the entire document input memory, thereby being not suit for streaming data processing. From the figure, we can see that QXSList and XQStream++ require relatively small size of memory, nearly constant amount. They process the incoming elements on the fly and remove elements form the buffer which will be no used in the future. This make the algorithm gained a good performance in memory usage and make it appropriate to process streaming data. In optimal QXSList, work stack is removed, and the buffer size is further reduced.



(a) QD1 over DBLP (b) QT1 over Treebank (c) QM1 over TMark

Figure 10. Comparison in Memory Usage

5. Conclusion

Streaming data are assumed to arrive continuously, and can potentially be unbounded. Processing streaming data query require algorithms process data in only consequential scan and return query results on the fly due to the limited storage space available especially when data is very large. In this paper, we proposed QXSList algorithm to process XML stream using list and relational pointer to maintain relationships between elements based on SAX parsing. This method use eager strategy, every element is visited only once, useless items can be removed in time, and target nodes are outputted on the fly. The experimental results show that this algorithm has considerable practical time and space performance advantages over other methods in massive volume and presence of deep recursion data sets.

As the future work, we will focus on two aspects, one is how to optimize this algorithm in reducing pointers numbers, because this is a main time cost, and the other is adopting some parallel strategy to improve efficiency.

Acknowledgments

This work is partly supported by Science and Technology Planning Project of Hebei Province (NO.15210126), and Science and Technology Research and Development Project of Langfang (NO.2015011066).

References

[1] J. Clark, and S. Derose, "XML Path Language (XPath) Version 1.0", http://www.w3.org/TR/xpath/, **(1999)**.
[2] X. Wu and D. Theodoratos, "A survey on XML streaming evaluation techniques", The VLDB Journal, vol. 22, no. 2, **(2013)**.
[3] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "Path sharing and predicate evaluation for high-performance XML filtering", ACM Transactions on Database Systems (TODS), vol. 28, no. 4, **(2003)**.
[4] M. A. Nizar, G. S. Babu and P. S. Kumar, "SFilter: a simple and scalable filter for XML streams", Proceedings of the 15th International Conference on Management of Data (COMAD), Mysore, India, **(2009)**.
[5] F. Peng, and S. S. Chawathe, "XSQ: A streaming XPath engine", ACM Transactions on Database Systems (TODS), vol. 30, no. 2, **(2005)**.
[6] V. Josifovski, M. Fontoura, and A. Barta, "Querying XML streams", The VLDB Journal, vol. 14, no. 2, **(2005)**.
[7] Y. Chen, S. B. Davidson, and Y. Zheng, "An efficient XPath query processor for XML streams", Proceedings of the 22nd International Conference on Data Engineering (ICDE), Atlanta, GA, USA, **(2006)**.
[8] W. S. Han, H. Jiang, H. Ho, and Q. Li, "StreamTX: extracting tuples from streaming XML data", Proceedings of the VLDB Endowment, Auckland, New Zealand, **(2008)**.
[9] G. Gou, and R. Chirkova, "Efficient algorithms for evaluating XPath over streams", Proceedings of the 2007 ACM SIGMOD international conference on management of data, Beijing, China, **(2007)**.

[10] B. G. Ryu, J. W. Ha, and S. K. Lee, "XQStream++: Fast tuple extraction algorithm for streaming XML data", Information Sciences, vol. 314, **(2015)**.
[11] D. Brownell, and D. Megginson, "SAX: Simple API for XML", http://www.saxproject.org/, **(2004)**.
[12] B. Choi, "What are real DTDs like", Technical Reports (CIS), **(2002)**.
[13] P. Boncz, T. Grust, K. M. Van, S. Manegold, J. Rittinger, and J. Teubner, "MonetDB/XQuery: a fast XQuery processor powered by a relational engine", Proceedings of the 2006 ACM SIGMOD international conference on management of data, Chicago, Illinois, USA, **(2006)**.
[14] M. H. Kay, "Saxon", http://saxon.sourceforge.net/, **(2013)**.
[15] M. Ley, "Dblp xml records", http://dblp.uni-trier.de/xml/, **(2003)**.
[16] A. Taylor, M. Marcus, and B. Santorini, "The Penn treebank: an overview, Treebanks", Springer Netherlands, **(2003)**, pp. 5-22.
[17] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu and R. Busse, "XMark: A benchmark for XML data management", Proceedings of the 28th international conference on very large data bases, Hong Kong, China, **(2002)**.

Authors

He Zhixue, Ph.D. candidate of Beijing University of Technology, The current research interests include large databases and parallel algorithms, and in particular, parallel algorithms for managing large databases.

