Aggregate Query Processing of Streaming XML Data

Yaw-Huei Chen and Ming-Chi Ho
Department of Computer Science and Information Engineering
National Chiayi University
{ychen, s0920206}@mail.ncyu.edu.tw

Abstract

Many XML query processing techniques have been proposed for processing structural joins. These algorithms find all element sets that satisfy the query pattern efficiently. However, sometimes we only need an aggregate value of the final result. Simpler algorithms are needed for processing the aggregate operations. Using start and end tags and a chain of linked stacks, we propose new algorithms for processing aggregate queries. Since no extra index structure is used in our method, the algorithm is suitable for processing streaming XML data. In addition, the experimental results indicate that the performance of our method is better than the index-based approach.

Keywords: aggregation, XML, databases, query processing, data streams.

1. Introduction

Data transferred on the Internet are not just in the form of data files. Some of the data, such as financial tickers, on-line auctions, and sensor data, are transferred as continuous data streams. For these kinds of application, the data are continuously changing and traditional relational database management systems (RDBMSs) can no longer efficiently support them. New methods have been proposed for processing data streams [2] [3].

XML (eXtensible Markup Language) [18] has become a standard format for data representation and exchange on the Internet. XML uses a tree-structured model to store data. When querying XML data, patterns are specified to identify the structural relationships between the elements in the tree structure. For example, query path expression "//book//title" indicates that element *book* has a descendant element *title*. In the path expression, "//" represents ancestor-descendent relationship and "/" represents parent-child relationship [19]. In order to find all elements that satisfy the query, many strategies have been proposed to evaluate XML data against the pattern specified in the query [1] [4] [5] [11] [12] [15] [20].

One of the methods, holistic twig join algorithm [4], uses a numbering scheme to represent the position of element occurrences in the tree structure and a chain of linked stacks to obtain matches for the query pattern. For example, as shown in Figure 1a, a 3-tuple (LeftPos, RightPos, LevelNum) is used to encode the data. The query shown in Figure 1b has a pattern "A//B//C". Each node in the query pattern has a corresponding stack as shown in Figure 1c. Pointers are used to indicate the next element in the stream. Since C_1 points to B_2 and B_2 points to A_2, [A_2, B_2, C_1] is an answer. Since A_1 is below A_2 on stack S_A, [A_1, B_2, C_1] is an answer, too. In addition, [A_1, B_1, C_1] is also in the query results as shown in Figure 1d. Holistic twig join algorithm is an index-based XML query processing technique. Since it is a set-at-a-time strategy, the index-based method is not designed for processing streaming XML data.

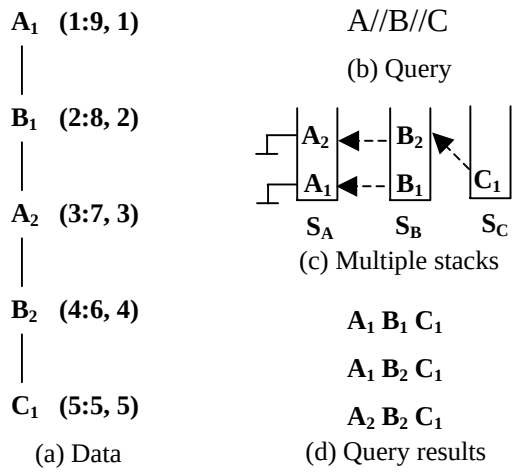


Figure 1. Compact encoding of answers using stacks.

Navigation-based algorithms compute query results by analyzing the input XML data one tag at a time. It has been shown that the index-based algorithm is more efficient than navigation-based algorithm if the required indices are already built [5]. Many navigation-based approaches have been proposed in the literature [6] [7] [8] [10] [14].

Most of the above-mentioned query processing techniques return the final results that satisfy the pattern specified in the query. However, in many situations we only want exact result size of a path expression instead of the final results. In other words, the aggregate value over the result set is desirable. An example of such aggregate queries is "count(//Book//Author)", which returns the number of "Author" under "Book" in the XML data input. The simplest method to evaluate an aggregate query is to find the final results for the query pattern first

and then apply the aggregate function to the resulting set. Obviously, this approach is inefficient.

A better performance can be achieved by using an XA-tree index structure, which indexes the aggregate value for each necessary XML element [13]. For example, in order to process the aggregate query "count(//Book//Author)", an XA-tree is built over all "Book" elements that contain at least one "Author" element. Then, the XA-tree is used to generate the resulting aggregate value. However, this approach needs to build and maintain the XA-tree index structure for necessary elements. This extra index structure causes the increases of the processing costs. Moreover, since the index structure needs to be built over all necessary elements, this approach is not suitable for processing streaming XML data.

In this paper, we propose a new method for processing aggregate queries of streaming XML data. Start and end tags are used to guide the process and linked stacks are used to record the count numbers of the necessary elements in the query pattern. We use the data stored temporally in the stacks to compute the aggregate values. Since no extra index structure is needed in our method, it is suitable for processing streaming XML data. In addition, the experimental results indicate that the performance of our method is better than the index-based approach.

The rest of the paper is organized as follows: Section 2 introduces the XML data model and aggregate queries. Section 3 presents the proposed method for processing aggregate queries of streaming XML data. Section 4 illustrates the experimental results. Section 5 contains concluding remarks.

```
//input: sequential XML data.
//output: aggregate value
  for each newnode from the SAX Parser {
    if (newnode is node) {
      if (newnode is start tag) {
        node's count + 1;
        if newnode is pat_end then CheckStack();
        if "average" then sum = sum / total;
      }
      else {
        if (node's count != 0) node's count – 1;
        else {
          pop node's count from node's stack;
          node's count – 1;
        }
        node = newnode;
      }
    }
    else {
      if (newnode is start tag) {
        if (node's count != 0) {
          push node's count to node's stack;
          if previous stack satisfies the pattern
            and is not empty then add pointer;
        }
        node's count = 0;
        node = newnode;   node's count + 1;
        if node is pat_end then CheckStack();
        if "average" then sum = sum / total;
      }
      else {
        node = newnode;
        pop node's count from node's stack;
        node's count – 1;
      }
    }
  }
```

Figure 2. Algorithm AggrStack.

2. Data model and aggregate queries

An XML document consists of a hierarchically nested structure of elements. Elements can be nested to any depth and the scope of an element is defined by a pair of start and end tags. An XML document can be treated as a rooted, ordered, and labeled tree, where each node represents an element. The XPath language [19] provides a way of specifying structural patterns that can be matched to nodes in the XML data tree. For example, "//Book//Author" represents the paths that contain an "Author" node as a descendent of a "Book" node in the XML data tree. Furthermore, paired start and end tags cannot interleave between each other, so that a sequence of tags like <A> <B> </A> </B> is not allowed in an XML document. Due to this characteristic, we can use start and end tags to guide the query evaluation process.

Count, minimum, maximum, summary, and average are common aggregate operations. Aggregate queries on XML data can be specified as *operator*(*pattern*), where *operator* represents one of the aggregate operations and *pattern* represents a path expression in XPath format. Conceptually, we first evaluate the path expression against the XML data and find all element sets that satisfy the query pattern. Then, the aggregate operator is applied to the resulting element sets to get the desired aggregation value. Count(pattern) computes the total number of resulting element sets that satisfy the query pattern; minimum(pattern) finds the minimum over the aggregate attribute values of all resulting element sets; maximum(pattern) finds the maximum over the aggregate attribute values of all resulting element sets; summary(pattern) computes the summary over the aggregate attribute values of all resulting element sets; and average(pattern) computes the average over the aggregate attribute values of all resulting element sets.

3. Aggregate query processing

Among the aggregate operations, count is an important one. We will present an algorithm for processing the count operation and discuss how to implement the other aggregate operations.

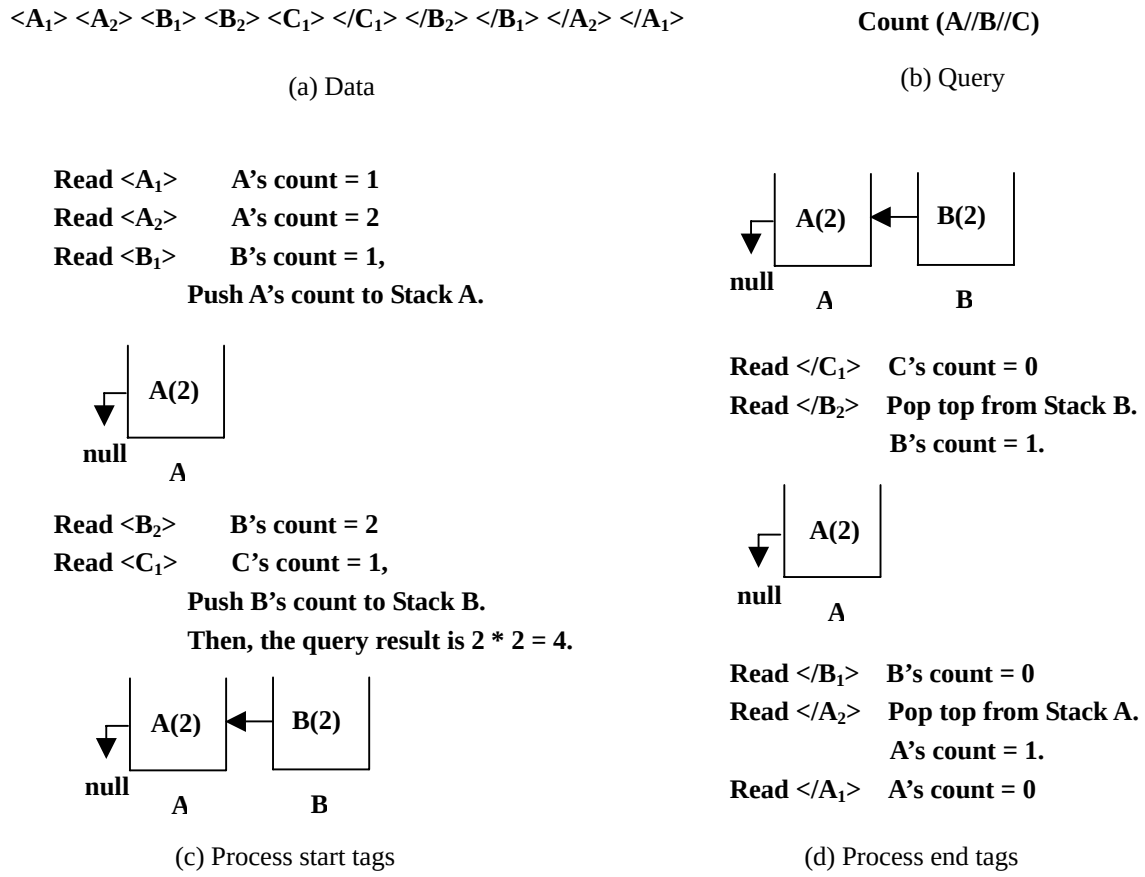


Figure 3. Compute count operation using stacks.

3.1. Count operation

The count operation is a fundamental aggregate operation. It can be expressed as count(pattern), in which the pattern is an XPath expression. In order to process this operation, we need to find out the number of element sets that satisfy the query pattern. However, because the actual structural join results are not required in the answer, we may use simplified structural join algorithms to get the aggregate value.

We use start tags and end tags (e.g., <A> </A>) in the XML data to direct the aggregate query processing. The input XML data are first processed by a SAX parser [16] and the resulting nodes are used as input data for our algorithm. As shown in Figure 2, algorithm AggrStack reads in nodes and checks whether the nodes can satisfy the query pattern. This algorithm is similar to the holistic twig algorithm in the sense of using a chain of linked stacks to store partial results. However, instead of storing structural relationships between elements in the stacks we store the count numbers of each necessary node in the corresponding stacks. More importantly, we do not use index to decide parent-child and ancestor-descendent relationships. Start tags and end tags are used to guide the pop or push operations of the stacks. Therefore, index structures are not necessary and more efficient processing can be achieved.

The basic idea of the algorithm is to store count information of each node in the corresponding stacks. We use pointer between nodes in consecutive stacks to keep track of a node's previous node in the query pattern. When we read in the start tag of the first node satisfied the query pattern (e.g., $<A_1>$), we increase the node's count number by 1. If the next qualified node is the same as the previous node (e.g., $<A_2>$, where the number 2 indicates that this node is the second occurrence of node A), we just increase the node's count number by 1. However, if the next qualified node is a new type node (e.g., $<B_1>$), we need to push the original node's count number information into its corresponding stack. If the node just pushed into the stack is not the first node in the query pattern, we need to add a pointer to the top node in its previous stack. When reading in an end tag (e.g., $</B_1>$), we decrease the node's count number by 1. If the count number is 0, we need to pop the node's count information from the corresponding stack and decrease it by 1. However, if the start tag of the node read is the last element in the query pattern, we need to check all stacks' count information and calculate and return the aggregate value.

Figure 3 illustrates an example of computing count operation using stacks. The input data consist of 5 elements, whose start and end tags are shown in Figure 3a. As shown in Figure 3b the query is count(A//B//C). Figure 3c depicts the process of

reading in the start tags of 2 A's, 2 B's and 1 C. Since C is the last element in the query pattern A//B//C, $<C_1>$ triggers the calculation of count aggregate value from the stacks. Figure 3d shows the process of reading in the end tags.

3.2. Other aggregate operations

Other aggregate operations will need different algorithms to compute. For example, when processing a minimum operation, we need to find the minimum value of an aggregate attribute in all element sets that satisfy the query pattern. The algorithm AggrStack can be modified to use current minimum value as an eliminating criterion to compute the minimum operation. All nodes that are not associated with the current minimum attribute in the stacks need to be eliminated from the stacks. Maximum operation can be implemented in a similar fashion. The other aggregate operations, summary and average, need to keep track of all aggregate attribute values and use the count operation to calculate the wanted aggregate values.

4. Experimental evaluation

In this section we evaluate the performance of algorithm AggrStack. Two sets of data, including a synthetic dataset and a real dataset, were used in the experiment. The AggrStack algorithm was compared with an index-based algorithm using the two data sets. We first introduce the experimental setup and then present the experimental results.

4.1. Experimental setup

The experiments were conducted on a machine with two 733 MHz Pentium III processors and 512 MB main memory running Windows 2003 Server. All algorithms were implemented in JAVA2 and Borland JBuilder 9. We used two XML datasets in the experiments. The first one was a synthetic XML dataset, which was generated by the XML Generator from IBM [9]. The DTD of the synthetic dataset is shown in Figure 4. The other dataset was a real application from the SIGMOD RECORD articles database [17]. The DTD of the real dataset is shown in Figure 5. Due to memory size limitation, the size of the datasets was about 500 KB. Although the datasets were not very large, we still could use them to evaluate the algorithms.

We compared our algorithm AggrStack with algorithm PathStack, that was modified from the holistic twig join algorithm proposed in [4]. XML documents were first parsed into nodes by SAX and then distributed into the corresponding stacks. Index values were also assigned to elements so that they could be processed by the PathStack algorithm.

4.2. Experimental results

```
<!-- <?xml encoding="US-ASCII"?> -->
<!ELEMENT nation (nation | country | city |
   company | department | manager | employee)+>
<!ELEMENT place (place | nation | country | city |
   company | department | manager | employee)+>
<!ELEMENT country (country | place | city |
   company | department | manager |employee)+>
<!ELEMENT city (city | country | place |
   company | department | manager | employee)+>
<!ELEMENT company (company | place |
   department | manager | employee)+>
<!ELEMENT manager (manager | employee |
   department | company | place)+>
<!ELEMENT department (department | manager |
   employee| company | place)+>
<!ELEMENT employee (name, email*, url*)>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT name (family, given)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT url EMPTY>
```

Figure 4. The DTD of the synthetic dataset.

```
<!ENTITY % carSet SYSTEM 'CarSet.cfg'>%carSet;
 <!ELEMENT SigmodRecord (issues)>
 <!ELEMENT issues (issue)*>
 <!ELEMENT issue (volume,number,articles)>
 <!ELEMENT volume (#PCDATA)>
 <!ELEMENT number (#PCDATA)>
 <!ELEMENT articles (article)*>
 <!ELEMENT article (title,authors)>
 <!ELEMENT title (#PCDATA)>
 <!ATTLIST title articleCode CDATA #IMPLIED>
 <!ELEMENT authors (author)*>
 <!ELEMENT author (#PCDATA)>
 <!ATTLIST author AuthorPosition
    CDATA   #IMPLIED>
```

Figure 5. The DTD of the real dataset.

We first conducted experiments on synthetic data to evaluate the algorithms. We used queries with path length from 3 to 9. For example, the length 4 query was "manager//department//employee//name". As shown in Figure 6, for all tested queries, the execution time for the count operation of our method AggrStack was less than both algorithms PathStack and PathStack+Index. In the implementation, PathStack could use existing index structures but PathStack+Index had to create index for the XML tree structures. Therefore, PathStack+Index took much longer time to process the same queries. Comparing PathStack and AggrStack, PathStack needed to distribute nodes into the corresponding stacks and used index values to obtain correct element sets and

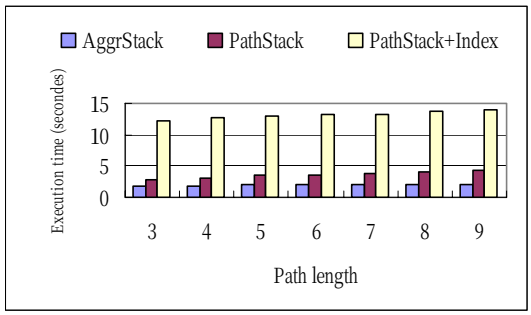


Figure 6. Execution time for synthetic datasets.

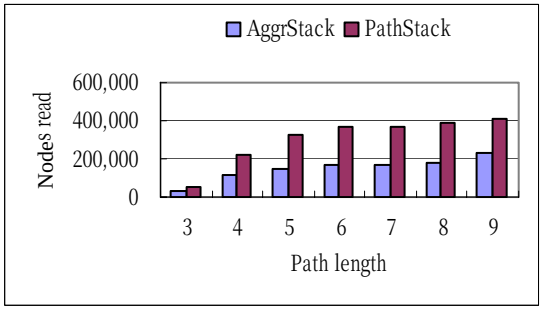
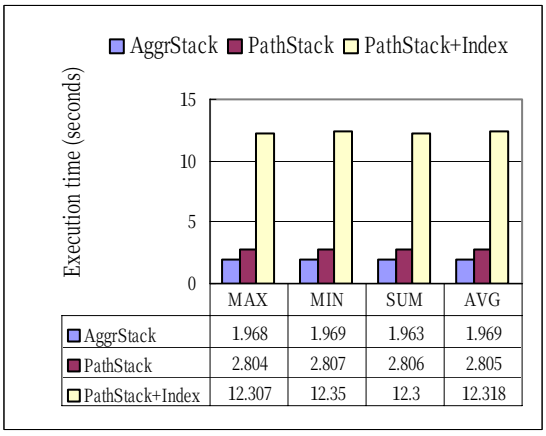


Figure 7. Nodes read for synthetic datasets.



	MAX	MIN	SUM	AVG
AggrStack	1.968	1.969	1.963	1.969
PathStack	2.804	2.807	2.806	2.805
PathStack+Index	12.307	12.35	12.3	12.318

Figure 8. Other operations on synthetic datasets.

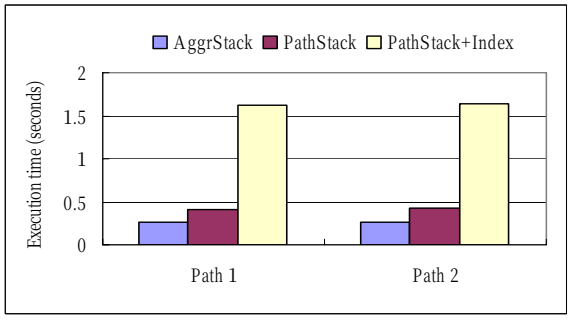


Figure 9. Execution time for real datasets.

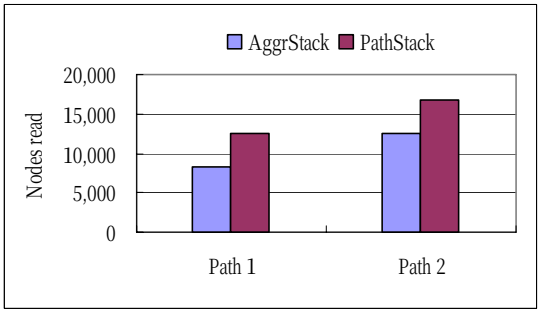
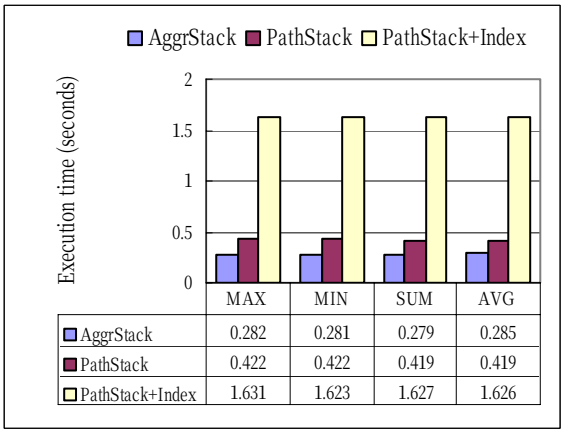


Figure 10. Nodes read for real datasets.



	MAX	MIN	SUM	AVG
AggrStack	0.282	0.281	0.279	0.285
PathStack	0.422	0.422	0.419	0.419
PathStack+Index	1.631	1.623	1.627	1.626

Figure 11. Other operations on real datasets.

then counted the aggregate values. Thus, AggrStack took less time to compute the count aggregate operation. Note that the difference of execution time was larger when the path length in the query was longer.

Figure 7 shows the number of nodes read from stacks by algorithms PathStack and AggrStack when processing the count aggregate operation. Algorithm AggrStack always read less number of nodes because the node information pushed into the stacks was only aggregate count values accumulated in the algorithm. On the other hand, algorithm PathStack pushed every individual node encountered into the stacks. Figure 8 indicates the execution time for the other aggregate operations on the synthetic datasets. Similarly, algorithm AggrStack had the least execution time among the three algorithms for all the other aggregate operations.

We processed two count queries in the experiments using real datasets of SIGMOD RECORD articles database. The path of the first query was "articles//authors//author" and the path of the second query was "issues//articles//authors//author". Figures 9 and 10 show the execution time and the number of nodes read for the operations, respectively. The execution time of other aggregate operations on the real datasets is shown in Figure 11.

In the real datasets, the same data did not repeat itself in the path. In addition, both paths of the queries used in the experiments were relatively short. Although algorithm AggrStack could not take advantage of storing aggregate count values in the stacks using the datasets, these experiments showed that AggrStack was still more efficient than PathStack. This is because PathStack needed to examine all stacks specified in the query but AggrStack could skip the "last" stack. Algorithm AggrStack did not need to handle the index structures as algorithm PathStack did was another reason that AggrStack was more efficient than PathStack. Since most streaming data do not have index structures already built in them, AggrStack is suitable for processing streaming XML data.

5. Conclusions

In this paper, we propose new techniques for processing aggregate operations, such as count, minimum, maximum, summary, and average, on streaming XML data. In order to efficiently process streaming XML data, we do not rely on index structures that are widely used in previously proposed XML query processing techniques. We use the start and end tags of each XML element to process the XML data. In addition, we use a chain of linked stacks to store intermediate aggregate count values so that the overall aggregate operations can be performed more efficiently. We have conducted a series of experiments using both synthetic and real datasets to evaluate the proposed techniques. The experimental results indicate that our method is more efficient than index-based aggregate query processing methods. While we present algorithms for a single path query, they can be extended to work for aggregation on more complex query patterns. Beyond that, aggregation on multiple XML data streams is another interesting research area.

Acknowledgements: This research is supported in part by the National Science Council under grant NSC93-2213-E-415-007.

References

[1] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," *Proc. of the 18th Int'l Conf. on Data Engineering (ICDE'02)*, February, 2002.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," *Proc. of 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 1-16, 2002.

[3] S. Babu and J. Widom, "Continuous Queries over Data Streams," *ACM SIGMOD Record*, vol. 30, issue 3, pp.109-120, September, 2001.

[4] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Join: Optimal XML Pattern Matching," *Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 310-321, 2002.

[5] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava, "Navigation- vs. Index-Based XML Multi-Query Processing," *Proc. of the 19th Int'l Conf. on Data Engineering (ICDE'03)*, March, 2003.

[6] C.Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions," *Proc. of the 18th Int'l Conf. on Data Engineering (ICDE'02)*, February, 2002.

[7] Y. Diao and M.J. Franklin, "High-Performance XML Filtering: An Overview of YFilter," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 26, number 1, pp. 41-48, March, 2003.

[8] 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*, vol. 28, issue 4, pp. 467-516, December, 2003.

[9] A.L. Diaz and D. Lovell, *XML Generator*, http://www.alphaworks.ibm.com/tech/xmlgenerator, 1999.

[10] T.J. Green, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata," *Proc. of the 9th Int'l Conf. on Database Theory*, pp. 173-189, January, 2003.

[11] R. Kaushik, R. Krishnamurthy, J.F. Naughton, and R. Ramakrishnan, "On the Integration of Structure Indexes and Inverted Lists," *Proc. of the 2004 ACM SIGMOD Int'l Conf. Management of Data*, pp. 779-790, 2004.

[12] F. Lam, W.M. Shui, D.K. Fisher, and R.K. Wong, "Skipping Strategies for Efficient Structural Joins," *Proc. of the 9th Int'l Conf. on Database Systems for Advanced Applications (DASFAA 2004)*, pp. 196-207, March, 2004.

[13] K. Liu and F.H. Lochovsky, "Efficient Computation of Aggregate Structural Joins," *Proc. of the 4th Int'l Conf. on Web Information Systems Engineering (WISE'03)*, December, 2003.

[14] F. Peng and S.S. Chawathe, "XPath Queries on Streaming Data," *Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 431-442, 2003.

[15] P. Rao and B. Moon, "PRIX: Indexing And Querying XML Using Prufer Sequences," *Proc. of the 20th Inter. Conf. on Data Engineering (ICDE'04)*, March, 2004.

[16] SAX 2.0, http://www.saxproject.org/, 2002.

[17] SIGMOD RECORD articles database, http://www.acm.org/sigs/sigmod/record/xml/, 2002.

[18] W3C Recommendation, "Extensible Markup Language (XML) 1.0 (Third Edition)", http://www.w3.org/TR/REC-xml/, 2004.

[19] W3C Recommendation, "XML Path Language (XPath) Version 1.0", http://www.w3.org/TR/xpath, 1999.

[20] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems," *Proc. of the 2001 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 425-436, 2001.