Real-Time Traffic-Data Analysis

Sudarshan S. Chawathe

***Abstract*— We describe a method for real-time monitoring of data generated by traffic sensors. We provide a system architecture and discuss three key components: (1) a streaming query processor that is used to reduce the volume of data; (2) a pattern-matching module that is used to detect when a developing traffic situation resembles one flagged earlier; and (3) an interface that efficiently displays the stream of sensor data in a user-configurable manner.**

I. Introduction

We propose a method for effective real-time use of the large volume of data that is generated by traffic sensors. Such sensors are increasingly common and measure variables such as the number of cars passing through a section of a lane, the weight of vehicles crossing a bridge, and the number of traffic signal violations or close calls. In addition, sensors that measure environmental conditions, such as temperature and rainfall, are also relevant. A judicious use of this sensor data, in conjunction with a knowledge of problematic patterns and situations from the past, allows a human expert to better monitor traffic in sensitive areas such as border checkpoints, stadiums, and tourist attractions. While offline analysis of such data is useful, greater benefits may be achieved by online, real-time analysis, which can help an expert to detect and plan a timely response to a developing public safety problem before the problem gets out of hand. For example, such analysis may detect a pattern of vehicles, parked near strategic intersections, that begin moving at approximately the same time.

Architecture Figure 1 outlines the system architecture. On the left, data from a large number (hundreds or thousands) of sensors of various kinds is aggregated and organized into a streaming form amenable to querying and analysis. For example, data from an electromagnetic sensor that detects passing vehicles but is identified using only a cryptic key may be combined with data mapping keys to intelligible names, such as *18th at M, northbound*. Methods for such aggregation and organization of sensor data are not the focus of this paper and have been discussed elsewhere [25]. We focus on the part of the system to the right of the vertical dotted line in Figure 1. We use XML as the common data format in this part of the system. Unlike the sensor network to the left of the dotted line, the modules to its right run on machines with substantial processing, energy, and network resources. Therefore, the space inefficiency of the XML format does not pose any serious problems, and is compensated for by the increase in usability.

As depicted in Figure 1, the system consists of three main components: 1) The *Streaming Query Processor* is used to select the data of interest from the incoming XML stream. The specification uses the *XPath* query language [9]. 2) The *Stream Viewer* is used to present a concise and configurable summary of the traffic data to a human analyst. It constitutes the main user interface and is also used to collect analyst input for refining the system's behavior. 3) The *Pattern Matching and Mining* module is used to detect user-specified patterns in the incoming stream and to mine historical traffic data for recurring patterns. We describe these three modules in order below, in Sections II–IV, and discuss related work in Section V.

II. Querying using XSQ

As depicted in Figure 1, the Streaming Query Processor module operates on the aggregated stream from the sensor network. It has two outputs, both produced in streaming form. The first output is used to drive the interactive display through the Stream Viewer. This output is customized using implicit and explicit preferences from the user. The second output is used to select data that is stored for later analysis. This output is likely to be larger than the first because the user may wish to save for later data that is not necessarily of current interest. On the other hand, it may not be practical to store all the data emerging from the sensor network; thus, some culling is required.

We use XPath [9] as the language for specifying both output streams as a function of the input. Thus, this module is essentially a streaming XPath query engine with two output streams. One of the strengths of the XPath language is that it permits intuitive and precise specification of interesting data, thanks to features such as such as multiple predicates, closures (the `//` axis), subqueries, and reverse axes (parent, ancestor). However, these features also pose significant challenges for a streaming query engine. Our methods, implemented in the XSQ system, are based on a template-based translation of XPath queries to automata with buffers [28]. These methods buffer data optimally in the following sense: At any point during query processing, any item that is buffered by XSQ must necessarily be buffered by any other streaming query engine.

Figure 2 depicts an example of one of the building blocks of the method used by XSQ: a pushdown transducer augmented with a buffer (BPDT). Each BPDT is responsible for evaluating a location step of an XPath query. The buffer is used for storing items that may be in the query result,

This work was supported by the National Science Foundation with grants IIS-9984296 (CAREER) and IIS-0081860 (ITR).

Computer Science Department, University of Maryland, College Park, MD 20742, USA. chaw@cs.umd.edu.

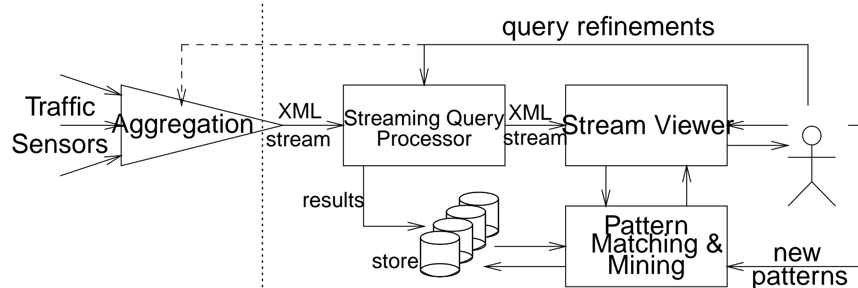

Fig. 1. System architecture for real-time monitoring of data from traffi c sensors.

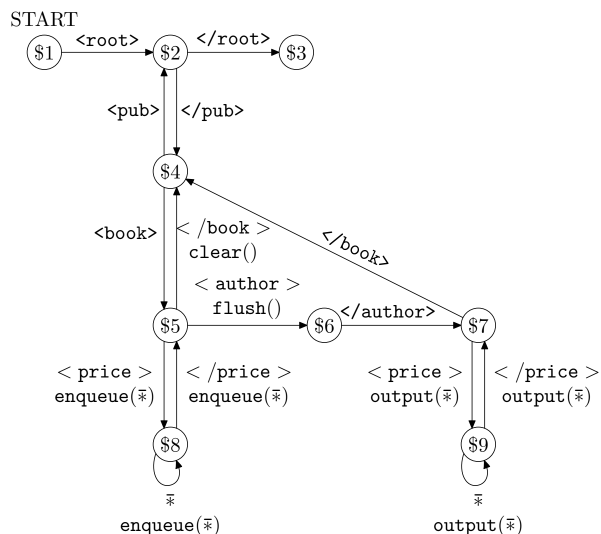

Fig. 2. A simple BPDT for query `/pub/book[author]/price`.

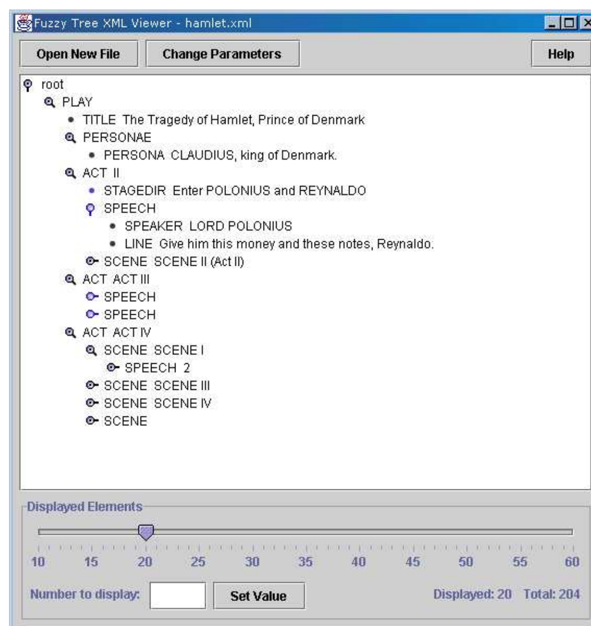

Fig. 5. Monitoring streaming data using the *FuzzyTree* stream viewer.

depending on the data that is encountered later in the stream. In Figure 2, when a *price* element is encountered in state 5, it is buffered using the enqueue operation. (The query specifies that only prices of books that have an author subelement are in the query result.) If an author element is encountered later, the transition to state 6 is accompanied by the flush operation, which sends the buffer contents to the output. Subsequent occurrences of price children of the current book element do not need to be buffered because an author element for this book is known to exist; thus such price elements are sent directly to the output from state 7.

In order to evaluate an XPath query composed of several location steps, XSQ uses a hierarchical arrangement of BPDTs, such as the one depicted in Figure 3. There are two key ideas. First, in any state, the hierarchical arrangement encodes the predicates in the query that are known to be true: A predicate is known to be true in the current state if and only if the current state is in the left subtree of that predicate's BPDT. For example, in state 15, the `[author]` predicate is known to be true (for the enclosing book) while the `[year>2000]` predicate has not yet been satisfied (for the enclosing `pub` element). Second, when control moves from a BPDT to its parent in the hierarchy, its buffer items are transferred to the parent's buffer using the upload operation. For details, we refer the reader to a longer paper [28].

III. MONITORING THE STREAM WITH *FuzzyTree*

Perhaps the most obvious method for monitoring a stream is displaying the last w elements encountered in the stream, where w is a parameter called the window-size. Although this method is easy to implement and is commonly used, it is not suitable for a high-throughput stream, such as the one generated by the traffic sensors in our architecture, because the list of the last w elements changes too rapidly to permit human comprehension or interaction. A simple alternative is refreshing the display only infrequently, in effect sampling the input stream periodically, at a low frequency. Although this method permits user interaction, it suffers from the obvious disadvantage of ignoring the vast majority of data in a high-throughput stream.

The general problem here is that of concisely summarizing streaming data. Our solution is based on ranking the XML elements that have been encountered in the stream using a score that has structural and user-defined

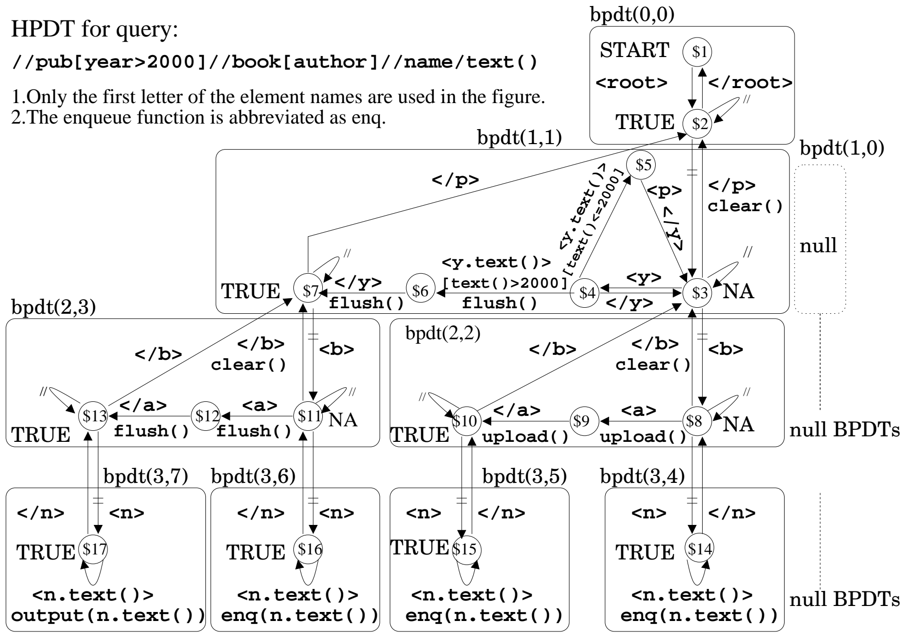

Fig. 3. HPDT: a hierarchical arrangement of BPDTs.

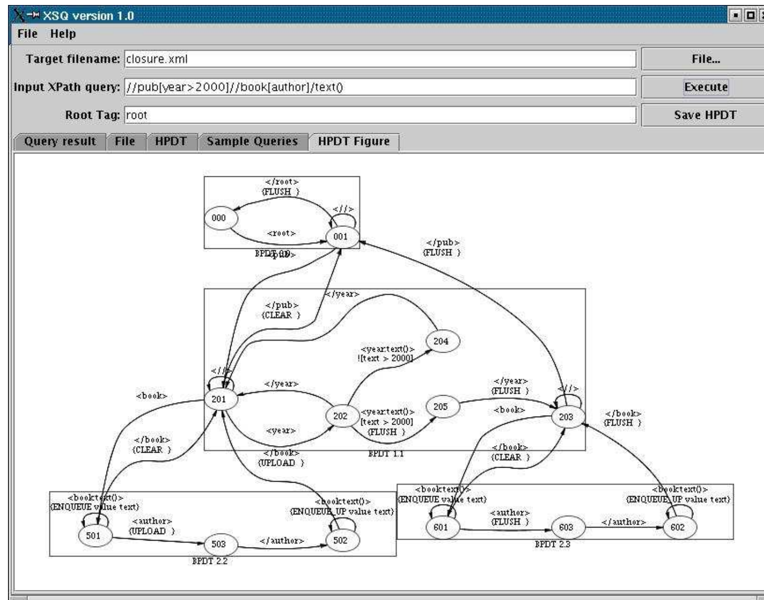

Fig. 4. Reducing the stream volume using *XSQ*.

components (described below). At any time, the w highest-scoring nodes are displayed. Here, w is a parameter that indicates the maximum number of elements that a user wishes to view simultaneously.

The structural component of a node's score intuitively favors nodes that participate in frequently-occurring patterns. In more detail, let t_i denote the tree (DOM [19]) representation of the XML data encountered before the position (equivalently, time) i in the stream. Let $t_i(n)$ denote the subtree rooted at node (XML element) n and let $|t_i(n)|$ denote the number of nodes in $t_i(n)$. Let $I_i(t)$ denote the set consisting of subtrees of t_i that are isomorphic to t. Then the structural score $s_i(n)$ of a node n at time i is defined as $s_i(n) = |t_i(n)| \times |I_i(n)|$.

The user-defined component of a node's score is based on explicit XPath specifications of interesting data as well as implicit specifications derived from the user's interaction with the displayed data. The implicit specification is derived by increasing the score of every displayed node on which the user clicks. In addition, the scores of nodes adjacent to a clicked node are also increased by amounts that decay exponentially with increasing distance from the clicked node. Each increase in a node's score resulting from such user interaction is in effect for a limited time,

h, representing the length of interaction history that is used for scoring. Limiting the effect of interactions in this manner is necessary to avoid confusing long-term effects of exploratory actions. The explicit specification is based on an optional set of XPath queries that the user may provide in order to fine-tune the display.

We have implemented these ideas in the *FuzzyTree* system. Figure 5 is a screen-shot of FuzzyTree displaying selected nodes from a dataset composed of XML versions of Shakespeare's plays [7]. The lower portion of the display is used to specify the parameter w using either the slider or the text input box. The system updates the display interactively in response, displaying the w nodes with the highest combined score. Other parameters, such as the history length h, are specified using the menus. The display of nodes (XML elements) is organized using a method similar to the commonly used *JTree* interface in Java [21].

One of the challenges in implementing FuzzyTree is the need to interactively update the display as the user-defined components of node scores change (as a result of explicit or implicit input from the user). The necessary operations, such as ranking nodes by isomorphism classes, are compute-intensive and would result in very poor performance if performed naively. Another challenge is the high rate of the input stream, which results in the XML tree changing rapidly.

IV. MINING WITH *SEuS*

The Pattern Matching and Mining (PMM) module (Figure 1) mines the stored data for *interesting* patterns. Patterns are deemed interesting if they satisfy some user-specified conditions (e.g., a cluster of vehicle sensors that report values greater than 20). A second source of interesting patterns is historical data in the store. For example, the PMM module may determine that a portion of the currently displayed data (in the Stream Viewer module) occurs very frequently in the store. Such detection of frequent patterns forms the basis of an intelligent system that can aid the analyst in predicting likely problems. This task may be addressed using a variety of methods, and we present only one here. (For example, current data may be matched with a database of incidents by computing nearest-neighbors in a high-dimensional space whose dimensions represent incident attributes [29].)

The problem of mining XML data for frequently occurring patterns may be formalized as follows: Given a labeled graph G (representing XML data), we wish to determine the set F of *frequent subgraphs*, which are defined as subgraphs G' that are isomorphic to at least s distinct subgraphs of G. The parameter s is a user-specified support level. For example, Figure 7 depicts a frequent subgraph for the data of Figure 6. Given the well-known hardness of the subgraph isomorphism problem, it is clear that an efficient solution to this problem requires a heuristic approach.

Our method, implemented in the *SEuS* system [14], is based on building the set F of frequent subgraphs by growing each of its elements one edge at a time. In order to prune the search space of subgraphs, we first compute a *summary* of the given graph. The summary is similar in spirit to *data guides* and other methods for extracting structure from graph data and provides an upper bound on the number of instances of any subgraph. Figure 8 depicts the summary for the data of Figure 7. When we grow the set F by adding, at each step, an edge to each of F's elements, we also compute the upper bound for each of the resulting subgraphs using the summary structure. If this bound is lower than s for some subgraph G'', we can safely remove from consideration G'' and all its descendants (all graphs that may be derived from G'' by adding one or more edges). (The descendants of a subgraph cannot be any more frequent than the subgraph.) Although (given the hardness of the problem) it is possible to devise inputs for which such pruning does not save much effort, our experiments indicate that for typical datasets this pruning is very effective [14].

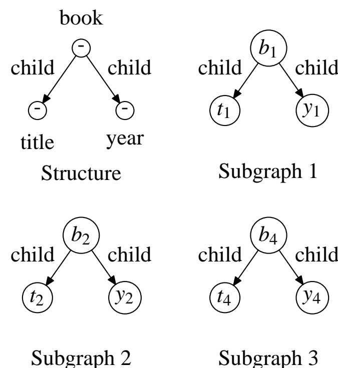

Fig. 7. A structure and its three instances for the data of Figure 6.

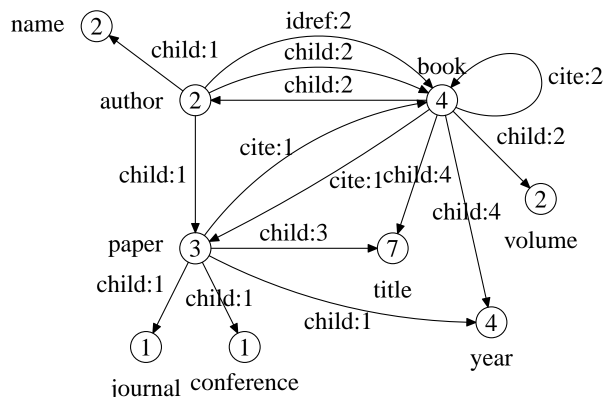

Fig. 8. Summary graph for the data of Figure 6.

V. RELATED WORK

XPath evaluation has received considerable recent attention. Most of this work focuses on the task of *filtering* documents using XPath expressions [2], [17], [13], [23], [8], [18]. In filtering, the input stream is assumed to be segmented into documents and XPath evaluation produces a boolean outcome for each document-query pair, indicating

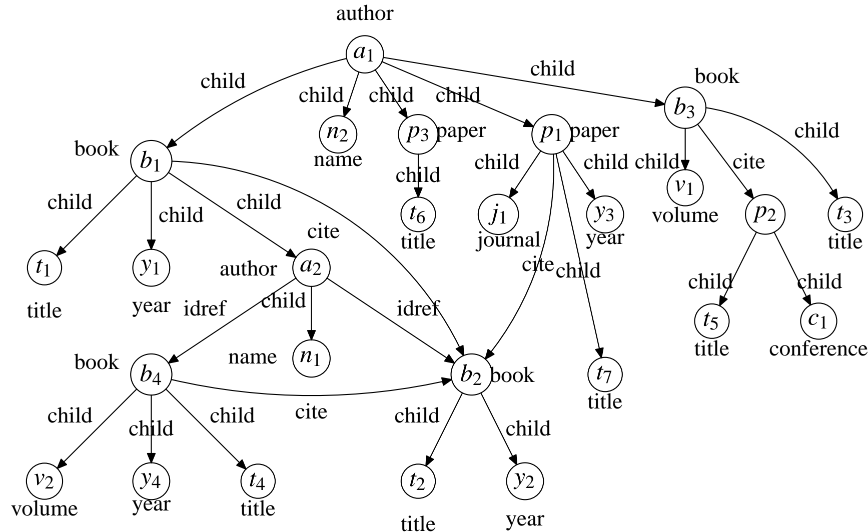

Fig. 6. Sample data for SEuS.

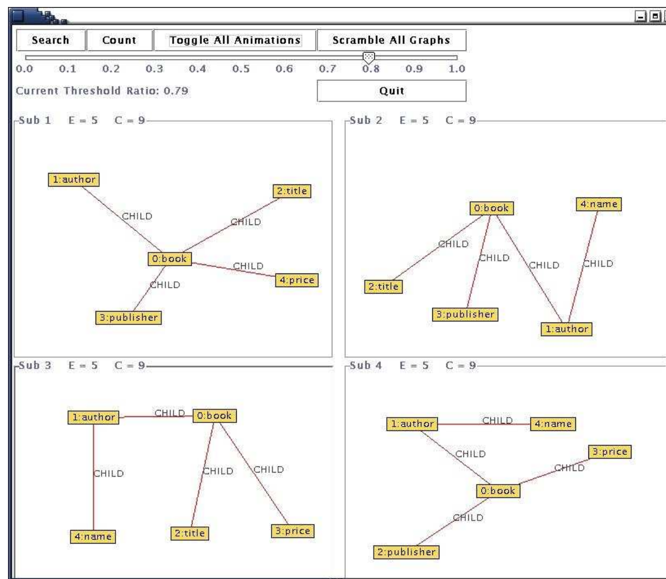

Fig. 9. Analyzing patterns using *SEuS*

whether the document contains data that matches the query. This task is simpler than the task of *querying* addressed by XSQ, which must evaluate an XPath query over an input that is not (necessarily) segmented into documents and determine the collection of elements that form the query result.

Among querying systems, the *XML Stream Machine (XSM)* [24] maps each subexpression of a decomposed XPath query to an automaton and uses a network of such automata for query evaluation. A similar method is used by *SPEX* [26]. In *XAOS* [4], [5], reverse axes are supported by filtering data using an *X-dag* data structure and storing potential results in an *X-tree* structure. However, query results are not generated until the end of the stream, when the X-tree is traversed. An alternate approach is to rewrite queries to replace reverse axes by forward axes [27].

Evaluating XPath in a non-streaming environment is known to be P-hard (combined data and query complexity) [16] and is amenable to polynomial-time main-memory algorithms [15]. The issue of expressiveness and containment for various fragments of XPath has also been addressed [6].

The method used by SEuS for analyzing patterns and discovering structure is applicable to any data that may be fruitfully modeled as a graph and is thus domain independent. Compared to the body of work on domain dependent methods, as surveyed by Conklin [11], there has been little work on domain independent methods for structure discovery. We discuss a few of these briefly: The CLiP system [30] guides its search for patterns using the estimated compression resulting from an efficient representation of instances of a substructure. Similarly, the SUBDUE [12] system uses a beam search guided by the

the minimum description-length principle, based on a fast, but inexact, graph-matching method as a subroutine. As in SEuS, structures in SUBDUE are generated by adding edges one at a time.

The FREQT algorithm [3] is limited to ordered trees, and is based on the idea of a rightmost expansion, which grows a tree by attaching nodes to its rightmost branch. The method by Cong et al. [10] is based on representing objects and patterns as a set of labeled paths with optional wildcards.

The AGM method [20] uses ideas similar to those used by the well-known *a priori* market basket analysis algorithm [1]: A $(k+1)$-itemset is a candidate frequent itemset only if all of its k-item subsets are frequent. In AGM, a graph of size $k+1$ is considered to be a candidate frequent structure only if all its subgraphs of size k are frequent. However, AGM searches for only frequent *induced* subgraphs. (Given a graph $G = (V, E)$, subgraph $G_s = (V', E')$ is called an induced subgraph if $V' \subset V$, $E' \subset E$ and $\forall u, v \in V', (u, v) \in E' \Leftrightarrow (u, v) \in E$.) The FSG system [22] uses a similar method, but is based on a sparse representation of the graph and other optimizations for candidate generation. Further, unlike AGM, FSG is not restricted to induced subgraphs. FSG and AGM are both based on a transaction database model in which the database graph consists of a collection of (relatively small) disjoint graphs representing transactions. While SEuS is applicable to such databases, it is also applicable to databases that cannot be decomposed into transactions in this manner.

REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th International Conference Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.

[2] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 53–64, Cairo, Egypt, September 2000.

[3] T. Asai, K. Abe, S. Kawasoe, et al. Efficient substructure discovery from large semi-structured data. In *Proc. of the Second SIAM International Conference on Data Mining*, 2002.

[4] C. Barton, P. Charles, M. Fontoura, D. Goyal, V. Josifovski, and M. Raghavachari. An Algorithm for Streaming XPath Processing with Forward and Backward Axes. In *The PLAN-X Workshop on Programming Language Technologies for XML*, Pittsburgh, PA, October 2002.

[5] C. M. Barton, P. G. Charles, D. Goyal, M. Raghavachari, V. Josifovski, and M. F. Fontoura. Streaming XPath Processing with Forward and Backward Axes. In *Proceedings of the International Conference on Data Engineering*, pages 455–466, Bangalore, India, March 2003.

[6] M. Benedikt, W. Fan, and G. Kuper. Structural Properties of XPath Fragments. In *Proceedings of the International Conference on Database Theory*, Siena, Italy, January 2003.

[7] J. Bosak. The plays of Shakespeare in XML. `http://www.oasis-open.org/cover/bosakShakespeare200.html`, July 1999.

[8] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the International Conference on Data Engineering*, pages 235–244, Feb. 2002.

[9] J. Clark and S. DeRose. XML path language (XPath) version 1.0. W3C Recommendation. `http://www.w3.org/`, Nov. 1999.

[10] G. Cong, L. Yi, B. Liu, and K. Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proc. of the Second SIAM International Conference on Data Mining*, 2002.

[11] D. Conklin. Structured concept discovery: Theory and methods. Technical Report 94-366, Queen's University, 1994.

[12] D. J. Cook and L. B. Holder. Graph-based data mining. *ISTA: Intelligent Systems & their applications*, 15, 2000.

[13] Y. Diao, P. Fischer, and M. J. Franklin. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of the International Conference on Data Engineering*, pages 341–344, San Jose, California, February 2002.

[14] S. Ghazizadeh and S. S. Chawathe. SEuS: Structure extraction using summaries. In S. Lange, K. Satoh, and C. H. Smith, editors, *Proceedings of the 5th International Conference on Discovery Science*, volume 2534 of *Lecture Notes in Computer Science (LNCS)*, pages 71–85, Lubeck, Germany, Nov. 2002. Springer-Verlag.

[15] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, Aug. 2002.

[16] G. Gottlob, C. Koch, and R. Pichler. The Complexity of XPath Query Evaluation. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 179–190, San Diego, California, June 2003.

[17] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with Deterministic Automata. In *Proceedings of the International Conference on Database Theory*, pages 173–189, Siena, Italy, January 2003.

[18] A. K. Gupta and D. Suciu. Streaming Processing of XPath Queries with Predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 419–430, San Diego, California, June 2003.

[19] A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) level 2 core specification, version 1.0. W3C Recommendation. `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113`, Nov. 2000.

[20] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 13–23, 2000.

[21] Java 2 platform, standard edition, v 1.4.2 API specification. Sun Microsystems. `http://java.sun.com/`, 2003.

[22] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of the 1st IEEE Conference on Data Mining*, 2001.

[23] L. V. Lakshmanan and P. Sailaja. On Efficient Matching of Streaming XML Documents and Queries. In *The 8th International Conference on Extending Database Technology*, pages 142–160, Prague, Czech Republic, March 2002.

[24] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 227–238, Hong Kong, China, August 2002.

[25] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2003.

[26] D. Olteanu, T. Kiesling, and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. Technical Report PMS-FB-2002-12, Institute for Computer Science, Ludwig-Maximilians University, Munich, May 2002.

[27] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management (XMLDM) at the 8th Conference on Extending Database Technology*, pages 109–127, Prague, Mar. 2002. Springer-Verlag.

[28] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 431–442, San Diego, California, June 2003.

[29] Y. Qi and B. Smith. Identifying nearest-neighbors in a large-scale incident data archive. In *Proceedings of the 83rd Annual Meeting of the Transportation Research Board*, Washington, DC, Jan. 2004.

[30] K. Yoshida, H. Motoda, and N. Indurkhya. Unifying learning methods by colored digraphs. In *Proc. of the International Workshop on Algorithmic Learning Theory*, volume 744, pages 342–355, 1993.