Method of Complex Event Processing over XML Streams

Tatsuki Matsuda
Hosei University
3-7-2 Kajinocho
Koganei, Tokyo, Japan
14t0011@cis.k.hosei.ac.jp

Yuki Uchida
Hosei University
3-7-2 Kajinocho
Koganei, Tokyo, Japan
13t0005@cis.k.hosei.ac.jp

Satoru Fujita
Hosei University
3-7-2 Kajinocho
Koganei, Tokyo, Japan
fujita_s@hosei.ac.jp

ABSTRACT

This paper describes a query processing engine for multiple continuous XML data streams with correlated data as a notification mechanism for navigating data exploration. Stream processing, including formal models for stream filtering, union, activation, decomposition, and partition, is formulated in algebraic expressions. In addition, a query language, called QLMXS, over XML streams for complex event processing is described. QLMXS supports all functions of the algebraic expressions in a SQL-like form. QLMXS queries are converted into a visibly pushdown automaton (VPA) that analyzes complex event data from the XML streams. The VPA engine concurrently processes multiple XML data on multiple levels; therefore, it is very important to tune the performance of the engine. Four optimization methods are proposed to improve performance by utilizing VPA and XML features: VPA-state reduction, VPA unification, delayed evaluation, and elimination of unnecessary XML processing. Experimental results demonstrate that VPA unification increases the processing speed of the VPA engine 1.6 times, and the overall processing speed is increased 2.6 times.

Categories and Subject Descriptors

H.2.3 [**Information Systems**]: DATABASE MANAGEMENT - Languages, Query languages

Keywords

Complex Event Processing, Stream Data Processing, Navigation, XML, Visibly Pushdown Automaton

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ExploreDB'15 May 31-June 04 2015, Melbourne, VIC, Australia
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3740-3/15/05 $15.00
DOI: http://dx.doi.org/10.1145/2795218.2795220.

1. INTRODUCTION

Exploration of databases and the web is one of hot topics in data processing area. Most of researchers interesting to this topic are focusing on methodologies how to express user's information needs in a smarter way, how to navigate users with interaction to find information around their goals, and how to present related items to widen the users' scopes. In such researches, internal issues in the exploration are mainly discussed, though external changes in surrounding environments also make some impacts to data exploration since the exploration is a time-consuming task so that users' interests might be changed during the exploration affected by the environments. This paper concerns a relationship between internal and external navigation in the data exploration. The external stuffs give a trigger to the users at the beginning of the exploration and continuously influence them during the internal exploration. Important factors in the external navigation are sensitivity to changes interesting to the users, immediate information delivery just on time, and relevance to the current interests in exploration.

Under such consideration, complex event processing (CEP), which is a method to find a combination of events coming from several information sources, plays an important role for the external navigation. Although there are many types of sources, this paper specially focus on XML streams since the XML format is widely used in the Internet communication. XML is a general purpose data format for representing hierarchically structured documents with tags. XML documents are used in many situations, such as system configuration files, data interchanging formats for Web services, and standard business documents. Currently, such XML formats are used in weather sensing broadcasts, stock information transfers, and GPS data communication so that these data are useful to apply to the external navigation in the exploration.

Several studies have examined high performance XML processing. For example, XSeq has been proposed to process time-dependent XML streams[4]. XSeq is an extension of XPath for representing patterns of correlated XML stream data and an analysis engine based on visibly pushdown automaton (VPA)[1].

This paper first describes requirements of the CEP for the external navigation in the exploration and then describes a way to build an efficient CEP engine which analyzes XML data in a stream. Currently, many data sources that send heterogeneous data continuously are available on the Internet. Therefore, this paper focuses on multiple stream processing in this context. A formal representation of multiple stream processing in algebraic expression is proposed, and a

query language for multiple XML streams (QLMXS) is designed to describe filtering conditions and formatting rules over multiple XML streams. Finally, a processing engine based on VPA is proposed and optimization techniques are introduced.

2. COMPLEX EVENT PROCESSING

CEP is a well-known technique for high speed analysis over continuous stream data. Many studies have examined stream analysis and effective use of CEP systems. Stream data have distinguishing characteristics that differentiate them from large scale databases, such as difficulty in predicting data arrival time, incremental data growth, and a variety of data formats. Therefore, stream data processing by the analysis engine should run only when specific conditions are fulfilled. In contrast to traditional analysis of data stored in a database, stream processing can output results at low latency so that it is possible to navigate user's exploration on time. In addition, in recent years, CEP engines have become able to process multiple streams because of the wide-spread use of sensor networks and a variety of Internet business activities. As a result, CEP engines must be able to handle more complex and combined analysis.

XSeq is a stream processing engine for XML streams that uses a query language that can retrieve and analyze sequential events from a data stream[4]. Since traditional query languages for XML, such as XPath and XQuery, are insufficient for representing patterns for CEP applications, XSeq extends XPath to support the Kleene-* operation and a '\' operator, which specifies the immediately next sibling of XML element. XSeq is effective for analyzing sequential XML streams; however, it cannot process multiple XML streams simultaneously. The XSeq engine uses VPA, which is a restricted pushdown automaton, as the basis of the processing engine. VPA stack operations are categorized into push, pop, and internal operations, and its competence is equivalent to finite state automata (FSA), i.e., VPA can be optimized as well as FSA[1]. VPA can model nested data structures, such as XML and JSON, with the stack operations such that it can process XML data efficiently. Because of these characteristics, VPA can also express and optimize XSeq queries effectively. In the XSeq engine, optimizations are applied to the VPA, such as shortening the prefixes, which are predictable from queries, removing the non-determinacy in VPA, and reducing the number of states. As a result, XSeq achieves high-performance processing over an XML stream.

Cayuga is another stream processing engine that attempts to process stream data from multiple sources[2]. Cayuga's query language provides a way to retrieve necessary data from multiple streams in an SQL-like form. In addition, Cayuga employs NEXT and FOLD operators to handle time-dependent causality between events in multiple stream data. Cayuga can produce a combined result from such dependent events. Cayuga also adopts a single queue architecture to avoid disordered data arrival when handling multiple streams. Note that this architecture guarantees the time dependency between the target events.

3. EXPLORATION AND MULTI-STREAM PROCESSING

This paper models an exploratory process as interaction among internal and external forces, and users. The internal forces are caused by navigation. Navigation system at least produces three types of forces, such as directing, widening and reminding. The directing leads the users toward their goals, the widening provides additional information related to their interests, and the reminding brings back historical searching memories to the users. On the other hand, the external force comes out of the system. At first, user's motivation of exploration is caused by the external force. Then unexpected events interrupt their exploration and sometimes change their directions. This paper focuses on a way to find important and relevant events from changes of surrounding states and designs a stream processing engine.

Three types of information are required for the streaming engine to capture, that is, a situation appearing at that time, a fusion of multiple situations appearing, and a time-sequential pattern of events. The first requirement for a stream engine corresponds to a notification mechanism for users when the engine detects changes of situation. Therefore, the engine must monitor and filter the data satisfied with a condition. The second requirement corresponds to generation of useful events by fusion of multiple events from multiple information sources. Then, the engine needs to implement functions which combine or partition streams, and have a scalable performance to process highly-parallel streams. The third requirement corresponds to detecting patterns of temporal change in any situations. The engine has to store past situations temporarily and find time dependency.

This section proposes a model in algebraic expressions for fundamental processing over multiple XML streams, satisfied with the above requirements of external forces.

(1) Filtering

Filtering represents a conditional selection from streams. For example, a process that applies a filter f to a stream s and outputs the result to a stream s' is expressed as follows.

$$s|f >> s' \tag{1}$$

(2) Union

Union joins two streams into a single stream. For example, a process that joins streams *s1* and *s2* into another stream *s3* is expressed as follows.

$$s1 + s2 >> s3 \tag{2}$$

(3) Activation

Activation represents a chain of events in which the first event triggers the second event. The output of the process is composed of data elements from both events. For example, there is a case wherein an event that occurs in stream *s1* activates the next event processing in stream *s2*, and the result is given to stream *s3*. This is expressed as follows.

$$s1 * s2 >> s3 \tag{3}$$

Note that this model implies order relation, i.e., it is not associative.

$$s1 * s2 \neq s2 * s1 \tag{4}$$

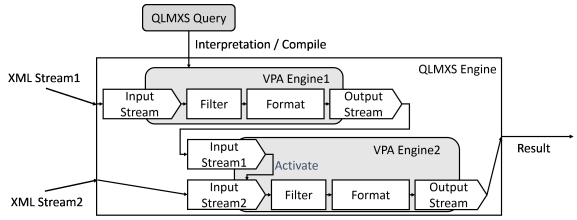

Figure 1: QLMXS Process Flow

(4) Decomposition

Decomposition is a process that splits stream data according to a specific key that corresponds to an element tag in XML. For example, a process that decomposes data from stream s to partial data by key k and outputs the results to stream s' is expressed as follows.

$$s/k >> s' \tag{5}$$

(5) Partition

Partition is a process that divides a stream into multiple streams as an array according to each specific key-value pair. For example, a process that partitions stream s into a stream array $s'[]$ by key k is expressed as follows.

$$s[k] >> s'[] \tag{6}$$

Extraction from the stream array according to a specific value is expressed as follows.

$$s'['value'] >> s'' \tag{7}$$

4. QLMXS

4.1 Overview of QLMXS System

We designed the QLMXS query language for multiple XML streams with reference to prior query languages. QLMXS is based on the model discussed in Section 3. QLMXS introduces additional unique expressions to achieve multiple stream, sequential, and loop processing. QLMXS queries are compiled into engines using VPA, which is suitable for XML processing, to analyze XML streams accurately and quickly. Compiling QLMXS queries into VPA is described in Section 5. Figure 1 shows a process flow of the QLMXS engine, in which several VPA engines are deployed to analyze multiple XML streams. In each VPA engine, input XML is initially filtered according to the conditions described by the QLMXS queries. Then, the engine formats the XML according to the format described by the queries. The processing results of these VPA engines are then sent to other VPA engines or clients.

4.2 Language Specifications of QLMXS

QLMXS adopts simplified XPath expressions to apply XML pattern matching to XML documents and adopts SQL-like expressions to write queries in a way that is similar to XSeq and Cayuga. A standard QLMXS query is composed of `from`, `select`, and `return` clauses. The `from` clause specifies the name of the input stream. The `select` clause specifies the output XML format of the result. The `return` clause specifies the name of the output stream.

In addition to these fundamental clauses, QLMXS supports optional clauses that express filtering conditions, multiple correlated input streams, and simple calculations. The following examples illustrate the use of these clauses and special writing styles in a virtual situation that analyzes stock quote streams.

```
Example 1.
   return MyStockStream1
   select stock/price
   from StockStream1
   where stock[price/text()>=100 and @name='A']
```

This query searches a stock quote with a price that is greater than 100 with company name `A` from `StockStream1`. In the `where` clause, filtering conditions are described in an XPath-like form. The result is a partial XML structure that begins from a `price` element. The output is written into a stream named `MyStockStream1`. This query corresponds to the formal model (1) in Section 3.

```
Example 2.
   return MyStockStream1
   select stock/price
   from ( select *
            from StockStream1
            where stock[@name='A'] )
   where stock[price/text()>=100]
```

The result of this query is the same as the result of Example 1. However, this query differs from Example 1 relative to filtering conditions. The `from` clause specifies an inner query that extracts a stock quote of company `A` from `StockStream1` in a nested description. The result of the inner query is filtered by the condition described in the `where` clause of the outer query. This nested description is useful for expressing combined queries without specifying intermediate stream names. This query corresponds to the following formal model.

$$(s|f1)|f2 >> s' \tag{8}$$

```
Example 3.
   return MyStockStream2
   select mystock[$1/stock, $2/stock]
   chaining StockStream3, StockStream4
   where [$1/stock/price/text()
                        =$2/stock/price/text()]
```

This query extracts a pair of stock quotes with the same price from two different streams. The `chaining` clause is used instead of the `from` clause to specify two interdependent streams. The special paths `$1` and `$2` are used in the `select` and `where` clauses to access data elements in the first and second input streams, respectively. A data arrival event in the first stream `StockStream3` activates the query processing in the second stream `StockStream4`. The final result is generated from a combination of the two streams with a specific parent tag `mystock`. This query corresponds to the following formal model.

$$(s1 * s2)|f >> s3 \tag{9}$$

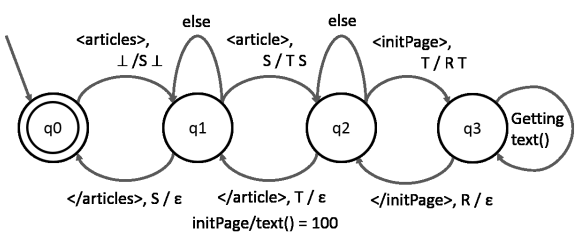

Figure 2: VPA Example

```
Example 4.
   return MyStockStream3
   select mystock[$1/stock, @ave=$sum div $cnt]
   chaining StockStream5, StockStream6
   setting $sum=0, $cnt=0
   processing $sum=$sum+$1/stock/price/text(),
                  $cnt=$cnt+1
   where [$1/stock/volume/text()
                        >=$2/stock/volume/text()]
   while 10 min
```

This query detects a duration wherein the trading volumes in the first stream are greater than the volumes in the second stream. The result involves the last data and the average price in the first stream within the duration. Note that the `setting` clause declares variables used in other clauses. In this query, variables `sum` and `cnt` are declared. The `processing` clause specifies the operations that run repeatedly when the conditions are satisfied. The processing of this query terminates when the conditions are not satisfied or the processing period is greater than the limit described in the `while` clause. This query corresponds to the formal model (9).

5. VPA GENERATION FROM QLMXS

As described in Section 2, the stack operations of VPA are categorized into push, pop, and internal operations such that VPA can be optimized easily and can model an interpretation of nested XML data structures. Therefore, VPA is effective for real-time XML stream processing.

VPA operations are defined with states, stack, and transition functions that refer to input symbols and the top of the stack. Thus, such information must be extracted from a QLMXS query to generate a VPA engine. The nested data structures for pattern matching in QLMXS are represented by simplified XPath expressions, in which node names correspond to input symbols, drill-down search in element hierarchy corresponds to transition functions, and intermediate stages in search correspond to VPA states. In other words, push-transition functions are associated with open tag processing, pop-transition functions are associated with close tag processing, and internal-transition functions are associated with attribute and character processing. As a result, valid levels of XML can be interpreted. Figure 2 shows the VPA that corresponds to the following query.

```
Example 5.
   return MyArticleStream
   select articles/article
   from ArticleStream
   where articles/article[initPage/text()>100]
```

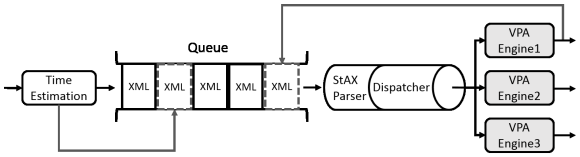

Figure 3: QLMXS Engine Architecture

This query attempts to find articles whose starting pages are greater than 100. In this query, it is obvious that `articles` and `article` tags and `article` and `initPage` tags have parent-child relations, respectively. The VPA states are generated by states after processing tag nodes in the XPath expressions of the query, in addition to the initial state. In Example 5, four states are generated, i.e., initial, post-`articles`, post-`article`, and post-`initPage` states. Here, transition functions are defined to connect these states for drill-down interpretation of XML. In addition, two `else` transition functions are defined to skip irrelevant tags.

The conditional predicates in XPath expressions must be handled in VPA operations to judge whether the conditions are satisfied. Such judgement should be performed just after encountering close tags. In Example 5, the filtering condition is evaluated after the `article` close tag arrives.

Activation plays an important role in detecting an event wherein correlated data come from multiple streams. Activation is implemented with a bridge between two VPA engines, such that satisfaction in the first engine triggers the second engine and satisfaction in the second engine produces the final result. Example 3 shows a typical case of activation in which data coming from `StockStream3` activates data processing from `StockStream4`. Here, assume respective VPAs for `StockStream3` and `StockStream4` are named VPA1 and VPA2, respectively. VPA2 remains inactive until the `stock` close tag arrives from `StockStream3` in VPA1. Then, the arrival of the close tag activates VPA2 and transfers the internal data of VPA1, such as `price`, to VPA2. Finally, VPA2 outputs the result into `MyStockStream2` in a form described by the `select` clause if the condition of `[$1/stock/price/text() = $2/stock/price/text()]` is satisfied.

6. IMPLEMENTATION OF QLMXS ENGINE

6.1 QLMXS Engine

Figure 3 shows the overall flow of QLMXS stream processing. As described in Section 5, the QLMXS engine processes XML streams with a combination of VPA-based analytical engines precompiled from QLMXS queries. The XML stream data are temporarily stored in a queue when they arrive to the engine, and the dispatcher transfers the data to the appropriate VPA engines in sequence. The result of each VPA engine might be re-sent to the queue if it is processed recursively by the VPA engines. Otherwise, it is output to the outer streams. The following descriptions explain each module in the QLMXS engine in further detail.

(1) Queue

The QLMXS engine attempts to process a series of XML data from multiple sources in correct order; therefore, strict time stamp management must be performed for all arriving

and intermediary data produced. Similar to Cayuga, the QLMXS engine achieves this time management using a single queue architecture, in which stream data are sorted by time stamp. If the arriving data do not have time stamps, the engine assigns a system time stamp before queuing. Note that the time stamps of intermediary data are identical to the original time stamps.

(2) XML Parser

The QLMXS engine uses the StAX parser to extract XML documents from the queue, divide them into elements, attributes, and text nodes, and send the nodes to the dispatcher. When multiple VPA engines require XML documents from the same stream, the StAX parser parses the XML documents once and sends the parsed data to the dispatcher, which then transfers the data to the appropriate VPA engines separately.

(3) Dispatcher

The dispatcher transfers the parsed data to the VPA engines according to a dispatch table in which the stream names are associated with the appropriate VPA IDs. Thus, the dispatcher can dispatch data to several VPAs.

(4) VPA Engines

Before initiating the QLMXS engine, QLMXS queries are interpreted and are compiled into the VPA engines. The engines sequentially process the XML data nodes received from the dispatcher. VPA processing uses many tokens, which have VPA stacks and move around VPA states to trace the transition functions when the XML nodes arrive. Therefore, the VPA engine runs slower as more tokens are used. In addition to standard VPA functions, the token has registers that cache the XML data for later use, such as data for conditional judgments and producing output. Note that a VPA can activate another VPA and transfer the cached data as required.

6.2 Optimization of VPA Engines

The performance of the VPA engines in the QLMXS engine affects the total processing time. Therefore, it is important to optimize the VPA engines.

6.2.1 *Reduction of VPA States*

For a case in which an XML schema is provided, a state generated from the path expression, which is not referred to in the filtering conditions or predictable from the subsequent states, is considered unnecessary and is removed from the VPA. This reduces the number of token operations in the corresponding states and achieves significant acceleration for large VPA.

6.2.2 *Unification of VPA*

If independent QLMXS queries have the same partial path expressions, the identical structures of VPA are duplicated in the independent VPA engines. In such a case, a new VPA engine is integrated to unify these structures for optimization. This reduces duplicate token operations and enhances the scalability of the VPA.

6.2.3 *Delayed Evaluation*

Note that the filtering conditions of QLMXS queries will not be satisfied if the nodes of the elements of the corresponding conditional evaluation do not arrive. Thus, the nodes are stored in a queue and token operations are suspended temporarily until the required node arrives. When the required node arrives, the nodes stored in the queue are popped and processed in a burst. If the node does not arrive, the stored nodes are discarded and the token operations are omitted. This optimization is effective for retrieval of optional elements in XML data.

6.2.4 *Elimination of Unnecessary Nodes*

Most of the search targets of the queries are located in limited parts of the XML documents. Therefore, eliminating unnecessary XML node processing reduces the number of token operations drastically. The QLMXS engine considers such unnecessary nodes as a sequence of characters without conversion to separated nodes such that the nodes are processed by a single token operation.

Table 1: Performance Evaluation

Number of data	Processing Time	Processing Speed
1000	87.4 ms	8.74×10^{-2}ms/data
2000	168.4 ms	8.42×10^{-2}ms/data
3000	281.4 ms	9.38×10^{-2}ms/data
4000	361.3 ms	9.03×10^{-2}ms/data

Table 2: Queries Set

Q1	/issue/articles/article/title/text()
Q2	/issue/articles/article/endPage/text()
Q3	/issue/articles/article[title/text() or endPage/text()]

7. EXPERIMENTS

This section evaluates the effects of the optimizations discussed in Section 6.2. The experiments use Intel(R) Core(TM) i5-2500K 3.3 GHz with 8 GB memory. The first experiment measures the primary performance of the QLMXS engine using the queries described in Example 3, in which data from two streams are compared in terms of price. Table 1 shows the results of processing times and speeds. As can be seen, the time increased linearly with the number of processed XML documents. The average processing time for a single XML document is approximately 9×10^{-2} ms. In other words, 11,000 data can be processed in one second.

The second experiment evaluates the effects of the proposed optimization methods. The target search document in this experiment is a SIGMOD Record[3], in which bibliographic items are stored. Table 2 shows the target queries. Here, queries Q1 and Q2 independently search documents from an identical stream, and query Q3, which is equivalent to the unification of Q1 and Q2, searches documents with two conditions simultaneously. Therefore, comparing the sum of execution performance of Q1 and Q2 with the execution performance of Q3 demonstrates the efficiency of the optimization discussed in Section 6.2.2. In addition, the optimizations discussed in Sections 6.2.1 and 6.2.4 are applied to Q3 to examine the efficiency of those optimizations.

Figure 4 shows the performance profiles against the data size for the optimized VPA engines, and Table 3 shows the number of token operations for each process. For processing of 467 KB of XML data, the processing time of Q3 is 38% less than that of the sum of Q1 and Q2. This de-

Table 3: Number of Token Operations (467 KB)

Q1+Q2	Q3	Q3(6.2.1)	Q3(6.2.4)
69518	34759	33188	15107

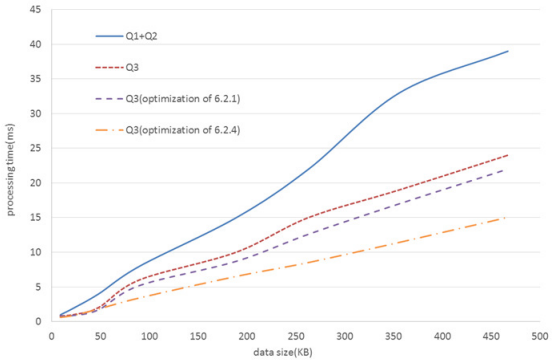

Figure 4: Experimental Results

crease in processing time results from the decreased number of token operations in common VPA states generated by `/issue/articles/article`. As can be seen in Table 3, the number of token operations in Q3 is approximately one-half of the sum of the number of token operations in Q1 and Q2. The processing times for the optimized VPA engines discussed in Sections 6.2.1 and 6.2.4 are 9% and 37% less than that of Q3, respectively. As described in Section 6.2.4, a more localized search target for the query results in increased unnecessary node processing. As a result, the efficiency of the elimination of unnecessary nodes is notable. As seen in Fig. 4 and Table 3, the number of token operations affects processing speed, and the differences in the processing times between the unoptimized and optimized engines increase linearly.

8. DISCUSSION

This paper discussed roles of stream processing in the data exploration. Three requirements were defined from a situation of the exploration, and then transferred to the algebraic representation of the stream processing. QLMXS is proposed as a query language over multiple XML streams.

Note that Cayuga achieves multiple stream processing using NEXT and FOLD operators. The NEXT operator expresses time-dependent events from multiple streams. When the second event occurs just after the first one, Cayuga delivers the events immediately. The NEXT operator cannot apply to an iteration of events occurring in several times. On the other hand, the FOLD operator keep monitoring until the specified condition is satisfied. QLMXS achieves similar capability by adopting the activation model discussed in Section 3. The `chaining` clause specifies the multiple target streams and the `while` clause specifies the expired time. In contrast to Cayuga's operators, QLMXS can distinguish the expressions of time-sequential processing and multiple streams processing clearly. Furthermore, compared to Cayuga, which was designed for general data processing, the QLMXS engine uses VPA to enable XML stream analysis.

XSeq has operators that represent a Kleene-* operation and the immediately next sibling elements in XML, and XSeq achieves complicated XML pattern matching using these operators. Conversely, the QLMXS engine is limited to simple pattern matching during detection, such as monotonically increasing and decreasing values. However, the processing speed of the QLMXS engine is faster than that of XSeq[4], because the QLMXS engine reduces the non-determinacy in VPA. In the QLMXS engine, several small VPA engines work together to process combinational queries, and the QLMXS engine achieves complex analysis across multiple data streams.

9. CONCLUSION

This paper has described a method to analyze XML data over multiple data streams as a notification mechanism for navigating data exploration. Algebraic expressions have been proposed for multiple stream processing. In addition, a query language, i.e., QLMXS, and an execution engine with several optimizations have also been proposed. The results of our experiments demonstrate that the proposed VPA optimizations result in significant acceleration.

In future, we plan to validate the proposed methods with more complex queries and implement acceleration by concurrent execution using a combination of QLMXS engines in distributed computing, such as MapReduce.

10. REFERENCES

[1] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211. ACM, 2004.

[2] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. M. White, et al. Cayuga: A general purpose event monitoring system. In *CIDR*, volume 1, pages 412–422, 2007.

[3] P. Merialdo. Acm sigmod record in xml. `http://www.cs.washington.edu/research/xmldatasets/data/sigmod-record/SigmodRecord.xml`. Accessed: 2015-02-27.

[4] B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over xml streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 253–264. ACM, 2012.