Enhancing applications with filtering of XML message streams

Kirill Belyaev and Indrakshi Ray
Computer Science Department
Colorado State University, Fort Collins, USA
{kirill | iray}@cs.colostate.edu

ABSTRACT

Modern applications often have to process and filter information in XML format for reasons of interoperability. Often times those XML messages arrive from publisher at unpredictable rates and must be processed in near real-time to answer complex filtering queries. Towards this end, we introduce Seshat – the content-based Domain Specific XML stream processing engine for meeting the needs of different subscribing applications. Seshat provides the full support for Boolean logic operators including negation and also supports supplemental operators, such as substring search. Its simple query framework enables filtering queries with variable substitution predicates. We describe the query processing engine and also the implementation details. Seshat engine can be potentially deployed in publish-subscribe brokers for selective message filtering and replication as well as in subscribing applications that need to process the arriving XML messages independently for the purpose of validation. We provide preliminary performance results of the filtering engine and its simple Domain Specific Language processing queries on several real-world XML datasets.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages; H.2.4 [**Database Management**]: Systems

Keywords

Content-based publish/subscribe, XML data/control streams

1. INTRODUCTION

Modern applications often have to process and filter information in XML format for reasons of interoperability. Often times those XML messages arrive from publisher at unpredictable rates and must be processed in near real-time to answer complex filtering queries. We refer to these as data stream flow processing (DSFP) applications. These flows are processed by an engine that is often implemented as a software library running/embedded within the application which is capable of timely processing and filtering large amounts of data as it flows from the peripheral to the center of the system. Some data streams adhere to proprietary streaming formats which limit their interoperability [7]. Interoperability is addressed to some extent by web applications where XML remains a de-facto inter-communication exchange format between Internet applications [10, 11]. Consequently, most of the publicly available data feeds are still exported in the form of XML documents and lately as JSON.

In recent years, the XML-based data (and possibly control) dissemination networks are starting to become a reality [10]. Most services [3, 11] disseminate the data to the end users who are responsible for filtering and processing the data. In such cases, the users' endpoints should have the capability to receive and process it. For some use cases, we would like to selectively deliver only those items in the data or control stream that the subscribing application needs and not saturate it with unnecessary data. For other use cases, we may provide the endpoint applications with all the data and expect them to retrieve whatever they need. We would also want applications to have the capability to process the data stream independently, in memory, on-the-fly, without the reliance on the stand-alone service for performing message post-processing. In short, we would like to build a general purpose filtering query engine that can be deployed by applications that require to selectively filter XML messages.

In this paper we introduce Seshat (named after the goddess of writing and knowledge in Egyptian mythology) – the content-based data stream flow processing engine for processing and filtering XML data streams in real-time at the granularity of individual messages. Seshat engine consists of an embeddable application library that allows content filtering of individual XML messages using filtering queries, expressed with Boolean logic, substring, and time operators, that are written in a simple Domain Specific Language (DSL). The language simplicity allows easy integration with XML consuming applications and obviates the knowledge of complex and tedious XPath/XQuery [5] semantics. The development is done in a modular fashion which makes it extensible to support new filtering query operators needed for emerging applications. Seshat engine does in-memory processing of XML streams which makes it efficient and suitable for real-time processing. The processing at the granularity of individual XML messages results in accuracy of filtering. Seshat could potentially be used in XML stream processing applications deployed over various nodes distributed over the network. For example, Seshat library can be utilized in

IDEAS '16, July 11-13, 2016, Montreal, QC, Canada

DOI: http://dx.doi.org/10.1145/2938503.2938509

publish/subscribe brokers for selective replication of XML data streams [3] and in policy orchestration for performing validations on published policy and configuration updates for complex distributed systems [4].

The rest of the paper is organized as follows. In Section 2 we provide some motivating examples of XML stream processing for consuming applications. The architecture of Seshat engine is given in Section 3. We present our prototype implementation and experimental results in Section 4. We enumerate some of the related works in Section 5. We conclude the paper and point to future research directions in Section 6.

2. APPLICATION-CENTRIC FILTERING

In this section, we present motivating examples that may have the need for the deployment of Seshat engine. In modern data services, information could be disseminated in XML format and is streamed to the subscribing applications via the network [11]. The XML stream may be transported via HTTP/HTTPS using application layer or directly streamed using raw TCP/IP socket connection [3]. In certain cases it is desirable for an application to have the capability to filter the data stream independently in memory on-the-fly without reliance on separate stand-alone service that will do post-processing of messages on its behalf. With this capability, the applications have complete flexibility to decide when and where the messages should be filtered. Hence, we refer to our approach as *application-centric* filtering which can be used in security and performance enhanced scenarios.

We now offer two applications which motivate the need for such flexibility and demonstrates how our engine can be used. The first is dissemination and selective replication of XML data streams of stock data. A sample filtering overlay operating on stock exchange semi-structured data stream could be constructed out of brokers that have message processing capabilities [3]. The selective replication of individual messages is carried out by the subscribing broker instances equipped with Seshat library. Brokers receive the XML stream from the disseminating root broker, register and process the distinct content filtering queries, and then send the filtered message stream to their child brokers and form a selective XML distribution network.

A second example where a similar architecture can be used is in XML based policy dissemination overlays. The Linux Policy Machine (LPM) [4] is used to specify and enforce policies [12] for data services deployed in "containerized" runtime environments where services are comprised of individual components isolated from each other. The policy updates pertaining to LPM instances running on various nodes are disseminated from a centralized policy server using XML messages. Every node requires different policies for "containerized" application services deployed on the node. Each policy has a corresponding node ID in the message as well as other access control related metadata to distinguish policies that belong to different service hosting nodes. In such a scenario, it is desirable from a security perspective that the LPM instances at the endpoints receive the configuration/policy stream and perform content-processing independently without reliance on the upstream broker. In the event of an attack on a policy server, malicious configuration/policy updates may be disseminated to the endpoint application. LPM equipped with Seshat library is able to verify/validate individual policy updates.

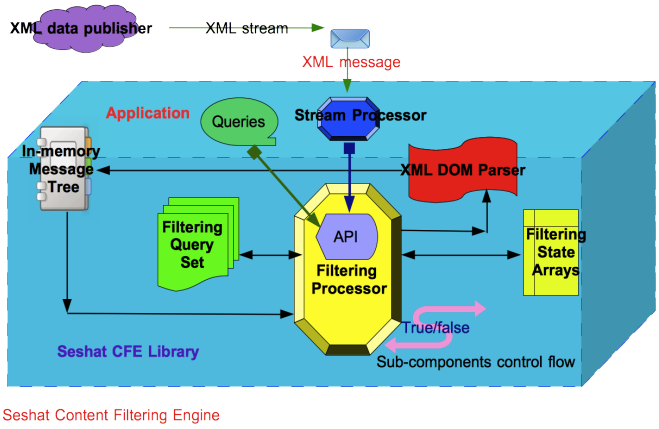

Figure 1: XML message processing

3. ARCHITECTURE

Our Seshat Content Filtering Engine (CFE) addresses the problem of efficient content filtering of individual XML messages. The XML messages are small in size (1 KB to 256 KB), which are typical for many Internet applications. The XML messages may be either data streams (as in the stock exchange example) or control streams (as in the case of policy update propagation) which may require processing using complex filtering conditions on the message elements. Our engine allows such complex filtering capabilities and is implemented in the form of Java application library that can be used by a wide array of Java applications that may need advanced XML content filtering capabilities.

We do not rely on the XPath or XQuery support for the purpose of content filtering for reasons of simplicity and efficiency. Moreover, XML query processing may require twig pattern matching [8, 9] which is not needed for the applications that we have in mind. In contrast to XPath, the developed Seshat query semantics allows the formulation of boolean logic expressions and other features that will be described later. We rely on tree-like DOM parsing that operates on the entire contents of the XML document and do not consider approaches based on Simple API for XML (SAX) that uses event-driven algorithms for parsing each XML document. Moreover, due to the modest message sizes (1KB to 256KB), the advantages of SAX over simpler DOM parsing with respect to memory utilization is insignificant [8].

XML message processing is depicted in Figure 1. Application-specific filtering queries are created by the component of an application or a separate service layer responsible for formulating such queries. Therefore, this task is not a part of Seshat functionality. The queries (labeled as Queries in Figure 1) are delivered by an application as text input to the Filtering Processor component. This task is accomplished via Application Programming Interface (API) that is provided by the Seshat library (labeled as API inside the Filtering Processor in Figure 1). Such API methods allow addition of intended queries, one query at a time. API also allows to delete all added queries. The added queries are stored in the Filtering Query Set (FQS) component. The XML message is received by application from the network stream

using application-specific Stream Processor component that is not a part of the Seshat library. Such a component delivers a message as the text input to the Filtering Processor. This task is accomplished via the API. Therefore, the library API allows two types of flow as input to the Filtering Processor: one is a control flow in the form of filtering queries and second is a data flow in the form of XML messages. The Filtering Processor invokes the XML DOM Parser component for the received XML message. XML DOM Parser creates the in-memory tree representation of the XML message. Filtering Processor, in turn, operates on the in-memory message tree and sequentially processes individual filtering queries which are stored in FQS component. The filtering state is maintained in Filtering State Arrays (FSA) component. FSA holds the following arrays: (i) *Tuples* – holds the subexpressions of the parsed query expression. (ii) *Elements* – holds the names of XML elements of the parsed query expression. (iii) *Logical Operators* – holds the logical operators of the parsed query expression. (iv) *Operators* – holds the operators (described in Table 2) on XML elements of the parsed query expression. (v) *Operands* – holds the operands of the operators – the values of XML elements of the parsed query expression.

An application may provide one or more queries, each of which may have one or more filtering conditions. A query with one filtering condition is known as an *atomic* query and that with more than one condition is known as a *composite* query. The multiple conditions of a composite query are connected via either a logical OR (|) operator or a logical AND (&) operator, but never both. Samples of filtering queries based on the NASDAQ stock quotes trading that operate on XML elements are shown in Table 1. For instance, Q1 is an example of an atomic filtering query and Q3 is an example of a composite query. The equality, negation and comparison operators may be used to evaluate string, integer, float and timestamp values depending on the operand type as per operator overloading properties. The relational operators supported by Seshat appear in Table 2. In addition, it also supports variable substitutions ($), such as, (AskRealtime = $BidRealtime).

Table 1: Sample Filtering Queries

Q1	Symbol = TDDD
Q2	Sector ! SEC0
Q3	Symbol = TDDD & Industry = IND00
Q4	Industry = IND00 & Year = 1999
Q5	Symbol = TDDD | Symbol = ACMR
Q6	Sector = SEC0 | Sector = SEC2
Q7	Industry = IND00 | Industry = IND01
Q8	CurrentPrice < 116.0 | OpenPrice > 177.0

Table 2: Relational Operators

Operator	Description	Example
=	equality	(Symbol = ABTE)
!	not-equal	(Symbol ! TDDD)
%	substring	(DividendPayDate % Dec)
<	relational less	(CurrentPrice < 116.0)
>	relational greater	(CurrentPrice > 116.0)

As noted, multiple queries submitted by the application are stored in FQS component and form a single FQS. Such a constructed set is evaluated based on the disjunctive normal form (DNF). In other words, a query set is a disjunction of atomic or/and composite queries. For instance, a single FQS may be constructed out of atomic and composite queries presented in Table 1. Any query from this set that evaluates to true would cause the engine to terminate further filtering and notify the invoking application via the output represented as a boolean true. If none of the registered queries evaluate to true, the engine will deliver false output to the components of the invoking application. Due to space limitation, we direct the interested reader to consult the sample unit tests for the engine workflow available at its GitHub repository [1].

4. EXPERIMENTAL RESULTS

In this section, we present the initial experimental evaluation of our Seshat CFE engine. The specification of the machine involved in the benchmarking is depicted in Table 3. Since Seshat is currently single-threaded, the benchmarking utilized only a single CPU core of the machine. The experiments have been conducted using a Java application that we term as the Dataset Reader. The Dataset Reader reads filtering queries from a separate text file and reads the XML dataset from the filesystem one message at a time. The Dataset Reader uses the Seshat CFE library to register and execute content-filtering queries on individual XML messages in a dataset. In our previous work [3], we have conducted a set of performance experiments on the content-based publish/subscribe broker, but did not assess the runtime information for the actual filtering engine.

Table 3: Node Specifications

Hardware	Description
CPU	Intel(R) Xeon (R) E5-2650 v2 @ 2.60 GHz
CPU Cores	8
RAM	32 GB
OS	Fedora 21, Linux kernel 4.1.13-100
Java VM	OpenJDK 64-Bit Server Java SE 8.0_65

Table 4: XML Datasets Information

Dataset	Message size	Count	Size
NASDAQ	0.47 KB	100 000	47 MB
Orders	0.57 KB	100 000	57 MB
S&P500	3.1 KB	10 000	31 MB
Christies Auction	54.36 KB	5 000	271 MB

The information on the datasets involved in the experiments is provided in Table 4. All datasets involved in the benchmarks are publicly available via our TeleScope XML stream broker at its GitHub repository [2]. Our benchmarking experiments show the various plots that refer to query sets consisting of 1, 4 and 8 queries respectively. In Table 5 we provide the sample FQS which corresponds to the S&P500 Companies XML dataset. We name the query sets after the number of queries contained in them. For example, FQS8 has eight queries as shown in Table 5. Individual queries in a set are separated by enclosed parentheses () and a comma for reasons of readability. Filtering queries in

Table 5 and Table 1 operate on elements of XML message (start with an upper case letter) and not on its attributes (start with a lower case letter). However, Seshat is capable of processing both types of nodes in XML document.

Table 5: Sample S&P500 Filtering Queries

[FQS1:] {(Symbol = ABT & FiftydayMovingAverage > $TwoHundreddayMovingAverage)}
[FQS2:] {(Symbol = ABT & FiftydayMovingAverage < $TwoHundreddayMovingAverage), (AverageDailyVolume > 900000 | Change_PercentChange = "+0.58 - +1.41%")}
[FQS4:] {(BidRealTime < $BookValue & FiftydayMovingAverage < $TwoHundreddayMovingAverage), (Change > 2.00 | AverageDailyVolume < 700000), (Symbol = AMAT | DaysHigh > 60.00), (Change < 0 & EarningsShare > 5.00)}
[FQS6:] {(BidRealTime > $BookValue & FiftydayMovingAverage < $TwoHundreddayMovingAverage), (Change > 3.00 | AverageDailyVolume < 1000000), (Symbol % AM | DaysHigh < 60.00), (Change > 0 & EarningsShare > 5.00), (StockExchange = NYSE | DividendYield > 3.00), (Change < -0.10 & DividendPayDate % Aug)}
[FQS8:] {(BidRealTime > $BookValue & FiftydayMovingAverage < $TwoHundreddayMovingAverage), (Change > 3.00 | AverageDailyVolume > 2000000), (Symbol % AM | DaysHigh < 60.00), (Change > 0 & EarningsShare < 5.00), (StockExchange = NasdaqNM | DividendYield > 3.00), (Change < -0.50 & DividendPayDate % Dec), (LastTradeDate = 10/3/2014 & EarningsShare > 5.00), (StockExchange = NYSE | PreviousClose < 40)}

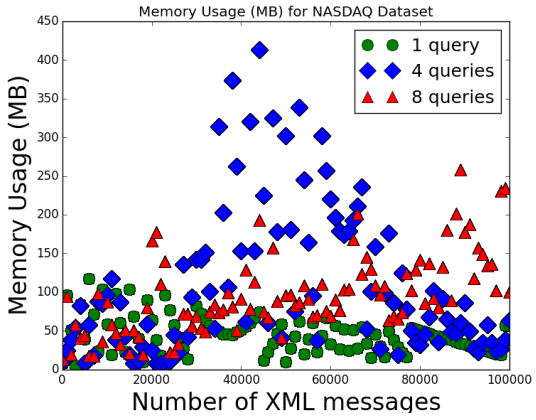

Figure 2: Memory Usage for NASDAQ Dataset

4.1 Memory Consumption

The following set of plots (Figures 2, 3, 4, 5) depicts the JVM memory consumption after the engine processes each message in the XML dataset. In general, the memory consumption does not increase or decrease uniformly based on the number of queries in the query sets. We believe that the garbage collection mechanism of JVM that reclaims memory according to its optimization strategies may account for such a behaviour. Note, that the Dataset Reader records the JVM memory utilization information every time it reads a new XML message from a dataset. That is reflected in the

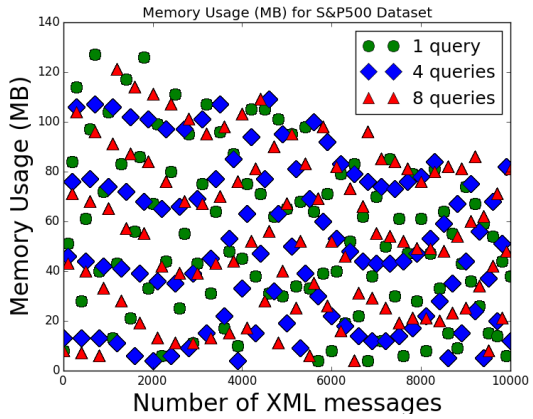

Figure 3: Memory Usage for S&P500 Dataset

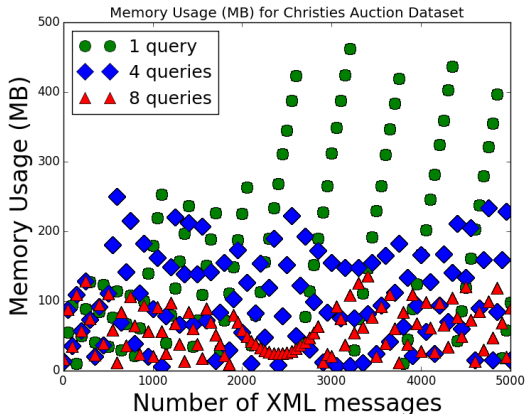

Figure 4: Memory Usage for Auction Dataset

dynamic behaviour of the memory utilization plots. For instance, Figure 2 for NASDAQ dataset shows that 8-queries set generally has higher memory utilization towards the end of experiment while 4-queries set has a utilization spike in the middle of the experiment. Memory consumption for S&P500 dataset in Figure 3 is relatively uniform for all filtering queries sets. For Christies Auction dataset in Figure 4 a single query set has the highest memory consumption, followed by 4-queries set while 8-queries set has the lowest memory utilization throughout the experiment. Memory usage for Orders dataset in Figure 5 shows that single query set has lowest utilization while 4-query and 8-query sets have a relative similarity with 8-query set having highest memory footprint from the beginning till the middle of the experiment. Orders dataset has the highest memory utilization although its individual XML message size is the second smallest after NASDAQ dataset as depicted in Table 4.

4.2 Filtering Time

Filtering time for various datasets using a single CPU core is depicted in Figures 6, 7, 8, 9. Both Figure 6 for NASDAQ dataset and Figure 9 for Orders dataset show similarity in filtering time for the 8-queries set. It is also the longest one. However, Figure 9 shows progressive increase in filtering time from a single to 8-queries set while Figure 6 does not

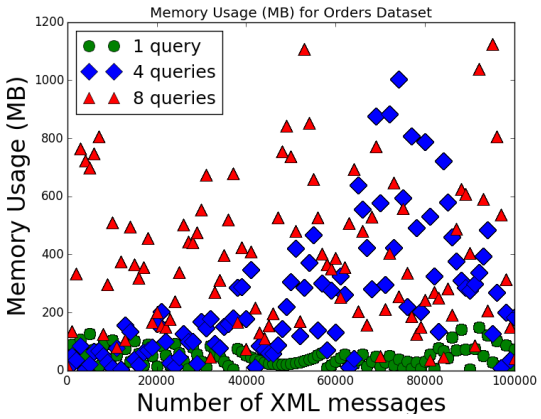

Figure 5: Memory Usage for Orders Dataset

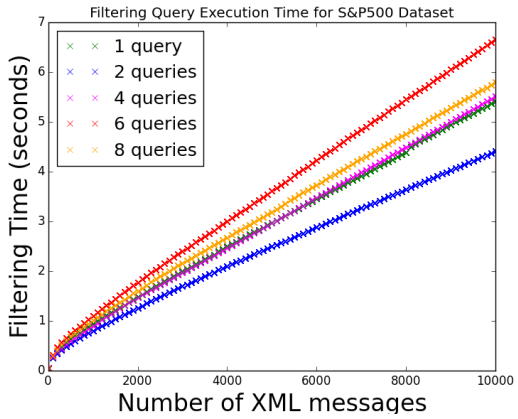

Figure 7: Filtering Time for S&P500 Dataset

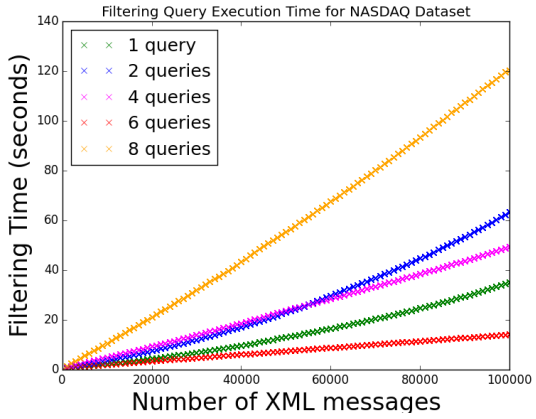

Figure 6: Filtering Time for NASDAQ Dataset

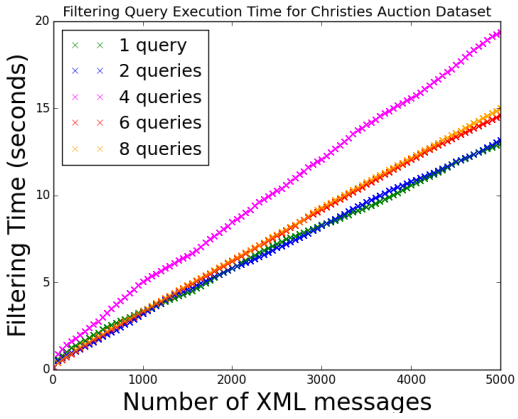

Figure 8: Filtering Time for Auction Dataset

show such a dependency. We can observe the size-based correlation – the sizes for both datasets are similar with 8-query set in Figure 9 taking slightly longer to filter due to larger dataset size. In Figure 7 for S&P500 dataset and Figure 8 for Christies Auction dataset we can observe that filtering time for 8-queries set is not significantly longer than that for a single query set. S&P500 dataset has the fastest filtering time among the 8-queries sets out of all four datasets. It is also the smallest in size as indicated in Table 4. Note, that the actual filtering queries for S&P500 dataset are provided in Table 5.

5. RELATED WORK

The problems of XML dissemination and efficient XML message processing in the context of content-based publish-subscribe paradigm have been extensively studied in the past [8]. Seshat is best suited for integration with content-based publish/subscribe (pub/sub) systems that are characterized by simple query languages, allowing simple selection predicates applied to individual events in a data stream. Research in content-based pub/sub systems often times emphasizes designing novel subscription languages and schemas [3].

In our earlier work on XML content filtering broker code-named TeleScope [3, 2], we have addressed the problem of efficient content filtering of individual XML messages arriving at high rates with the necessity for dissemination to a potentially large number of concurrent subscribers. Our current work on Seshat has incorporated and further enhanced the filtering capabilities of TeleScope filtering engine in the form of Java application library that has the potential to be used by a wide array of Java applications that may require advanced XML content filtering capabilities. Specifically, as covered in Section 3, Seshat offers a new set of filtering operators and new flexible query composition framework that offers improvement over the capabilities of the filtering engine originally introduced as part of TeleScope broker.

Content-based XML stream processing research in [13] [6] has closest resemblance to our work. However, in those filtering systems subscribers specified their interests in specific sub-parts of the XML document through XPath queries. The main goal of XML query processing is to find specific parts within XML document which match a query by building suitable indexes over it [8]. Therefore, the majority of XML filtering approaches use the twig patterns using XPath expressions to represent entities, such as user profiles [9]. We, on the other hand use pure content filtering, more concerned with finding specific filtering conditions that satisfy the entire contents of the XML message. Our work targets applications in which the subscribing data sinks need the entire contents of the messages [10]. Also, XQuery/XPath

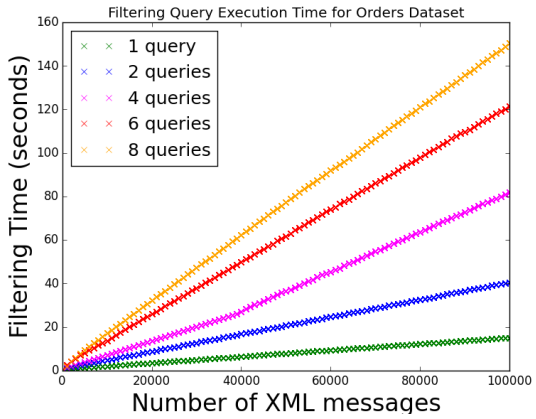

Figure 9: Filtering Time for Orders Dataset

based solutions such as YFilter [6] handle predicates on attributes or text data of elements which contain only equality operators and do not generally support boolean logic constructs such as AND/OR operators and negation expressions in value-based predicates [9].

Value-based predicate filtering of XML documents [9] using twig patterns provides limited expressiveness of XPath queries for efficient XML content filtering in designated application domains. The authors proposed the extension of XPath to build the comprehensive XML filtering system that evaluates value-based predicates in twig patterns and matches their structure holistically. Our solution does not currently rely on XPath and its twig patterns. Kwon et al. [9] uses the similar XML message access approach – the incoming XML documents that need to be filtered are initially parsed using a SAX parser. In our approach as discussed in Section 3 we use a DOM method that theoretically does not incur significant memory allocation overhead for short XML messages used in our deployment scenarios. Moreover, we require processing of entire XML document.

In contrast to most XML content filtering approaches, our work does not rely on XQuery or XPath and their subsets. We use the alternative tree-based approach that operates on the nodes of the XML message parsed into a DOM tree. Seshat language simplicity allows easy integration with XML consuming applications that does not require knowledge of complex and tedious XPath/XQuery semantics and detailed information of XML schema and positional location of element and attribute nodes beforehand for most of the common XML message content filtering scenarios. Our DSL language bears close resemblance with PADRES [7] query language used in the PADRES pub/sub system. In contrast to PADRES, our work targets XML message filtering and also supports negation and construction of filtering queries connected via boolean logic operators with support for variable substitution.

6. CONCLUSION AND FUTURE WORK

Many applications generate streaming data at various sources. In this work we have introduced Seshat – content filtering engine over XML streams that can be embedded within the stream processing applications to provide application-centric filtering capabilities. Such capabilities are desired in various types of applications, including efficient dissemination and selective replication for different categories of XML data streams, as illustrated in stock quotes example, and propagating validated policy updates across replicated servers. As part of future work, we plan to address the problem of inter-query sharing of processing results to increase the computational efficiency among a set of overlapping content filtering queries. We are also going to investigate the multithreaded implementation of the Seshat engine for simultaneous distribution of filtering load across a set of processing CPU cores.

7. ACKNOWLEDGMENTS

This work was supported in part by a grant from NIST under award no. 70NANB15H264.

8. REFERENCES

[1] K. Belyaev. Seshat - XML Content Filtering Engine Library. https://github.com/kirillbelyaev/seshat, 2016. accessed 12-May-2016.

[2] K. Belyaev. "TeleScope - XML Data Stream Broker/Replicator". https://github.com/kirillbelyaev/telescope, 2016. accessed 12-March-2016.

[3] K. Belyaev and I. Ray. Towards Efficient Dissemination and Filtering of XML Data Streams. In *Proc. of IEEE DASC*, pages 1870–1877. IEEE, 2015.

[4] K. Belyaev and I. Ray. Towards Access Control for Isolated Applications. In *Proc. of SECRYPT*, page to appear. SCITEPRESS, 2016.

[5] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML path language (XPATH). *World Wide Web Consortium (W3C)*, 2003.

[6] Y. Diao and M. Franklin. Query processing for high-volume XML message brokering. In *Proc. of VLDB - Volume 29*, pages 261–272. VLDB Endowment, 2003.

[7] H.-A. Jacobsen, A. K. Y. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. The PADRES Publish/Subscribe System. In *Principles and Applications of DEBS*. IGI Global, 2010.

[8] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *The VLDB Journal*, 14(2):197–210, 2005.

[9] J. Kwon, P. Rao, B. Moon, and S. Lee. Value-based predicate filtering of XML documents. *Elsevier DKE*, 67(1):51–73, 2008.

[10] G. Li, S. Hou, and H.-A. Jacobsen. Routing of XML and XPath queries in data dissemination networks. In *Proc. of ICDCS*, pages 627–638. IEEE, 2008.

[11] I. Miliaraki and M. Koubarakis. Foxtrot: Distributed structural and value XML filtering. *ACM TWEB*, 6(3):12, 2012.

[12] J. Singh, J. Bacon, and D. Eyers. Policy enforcement within emerging distributed, event-based systems. In *Proc. of ACM DEBS*, pages 246–255. ACM, 2014.

[13] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using XML. *ACM SIGOPS*, 35(5):160–173, 2001.