Emerald Insight

International Journal of Web Information Systems

Path-based keyword search over XML streams

Savong Bou Toshiyuki Amagasa Hiroyuki Kitagawa

Article information:

To cite this document:
Savong Bou Toshiyuki Amagasa Hiroyuki Kitagawa , (2015),"Path-based keyword search over XML streams", International Journal of Web Information Systems, Vol. 11 Iss 3 pp. 347 - 369
Permanent link to this document:
http://dx.doi.org/10.1108/IJWIS-04-2015-0013

Downloaded on: 01 November 2016, At: 22:54 (PT)
References: this document contains references to 30 other documents.
To copy this document: permissions@emeraldinsight.com
The fulltext of this document has been downloaded 59 times since 2015*

Users who downloaded this article also downloaded:

(2015),"Facet-value extraction scheme from textual contents in XML data", International Journal of Web Information Systems, Vol. 11 Iss 3 pp. 270-290 http://dx.doi.org/10.1108/IJWIS-04-2015-0012

(2015),"Raising resilience of web service dependent repository systems", International Journal of Web Information Systems, Vol. 11 Iss 3 pp. 327-346 http://dx.doi.org/10.1108/IJWIS-04-2015-0011

Access to this document was granted through an Emerald subscription provided by emerald-srm:563821 []

For Authors

If you would like to write for this, or any other Emerald publication, then please use our Emerald for Authors service information about how to choose which publication to write for and submission guidelines are available for all. Please visit www.emeraldinsight.com/authors for more information.

About Emerald www.emeraldinsight.com

Emerald is a global publisher linking research and practice to the benefit of society. The company manages a portfolio of more than 290 journals and over 2,350 books and book series volumes, as well as providing an extensive range of online products and additional customer resources and services.

Emerald is both COUNTER 4 and TRANSFER compliant. The organization is a partner of the Committee on Publication Ethics (COPE) and also works with Portico and the LOCKSS initiative for digital archive preservation.

*Related content and download information correct at time of download.

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

The current issue and full text archive of this journal is available on Emerald Insight at:
www.emeraldinsight.com/1744-0084.htm

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

Path-based keyword search over XML streams

Savong Bou
Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Japan, and
Toshiyuki Amagasa and Hiroyuki Kitagawa
Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Japan

Received 6 April 2015
Revised 17 April 2015
Accepted 18 April 2015

Abstract

Purpose – In purpose of this paper is to propose a novel scheme to process XPath-based keyword search over Extensible Markup Language (XML) streams, where one can specify query keywords and XPath-based filtering conditions at the same time. Experimental results prove that our proposed scheme can efficiently and practically process XPath-based keyword search over XML streams.

Design/methodology/approach – To allow XPath-based keyword search over XML streams, it was attempted to integrate YFilter (Diao *et al.*, 2003) with CKStream (Hummel *et al.*, 2011). More precisely, the nondeterministic finite automation (NFA) of YFilter is extended so that keyword matching at text nodes is supported. Next, the stack data structure is modified by integrating set of NFA states in YFilter with bitmaps generated from set of keyword queries in CKStream.

Findings – Extensive experiments were conducted using both synthetic and real data set to show the effectiveness of the proposed method. The experimental results showed that the accuracy of the proposed method was better than the baseline method (CKStream), while it consumed less memory. Moreover, the proposed scheme showed good scalability with respect to the number of queries.

Originality/value – Due to the rapid diffusion of XML streams, the demand for querying such information is also growing. In such a situation, the ability to query by combining XPath and keyword search is important, because it is easy to use, but powerful means to query XML streams. However, none of existing works has addressed this issue. This work is to cope with this problem by combining an existing XPath-based YFilter and a keyword-search-based CKStream for XML streams to enable XPath-based keyword search.

Keywords Web search and information extraction, Applications of web mining and searching, Indexing and retrieval of XML data

Paper type Research paper

1. Introduction

Extensible Markup Language (XML) (Bray *et al.*, 2008) is a popular and standardized markup language for semi-structured data, and has been widely used in many applications due to its simplicity and versatility. Hence the amount of data being exchanged in the form of XML has been growing.

This research was partly supported by the Grant-in-Aid for Scientific Research (B) (#26280037) and the program Research and Development on Real World Big Data Integration and Analysis of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

Due to the rapid growth of XML data, there are growing demands to exchange XML data as streams of data in real-time. Such type of XML data is called *XML streams* (Gupta *et al.*, 2004). Typical examples of XML streams are Web services, sensor networks, etc., and, in many applications, XPath is used to filter out unnecessary parts (Gupta *et al.*, 2004).

In real applications, one needs to deal with different XML streams generated from different information sources. Such data are often dissimilar in structure. When querying or filtering such data, traditional XPath is not always useful, because users need to know the schemas of XML streams being processed (Gupta *et al.*, 2004). Moreover, to formulate queries appropriately, users need to be familiar with the syntax of one or more query languages, such as XPath and XQuery, which is often a barrier particularly for novice users.

To such a problem, keyword search (Cohen *et al.*, 2003; Gawinecki *et al.*, 2008; Hristidis *et al.*, 2006; Hummel *et al.*, 2011; Shao *et al.*, 2003; Vagena *et al.*, 2008; Xu *et al.*, 2005), where only keywords are used to formulate queries, offers a simple, but effective mean. For example, in the sample XML data in Figure 1, let us assume that one wishes to retrieve information related to books about "Star Wars". In this case, he only needs to put three keywords, "book", "Star" and "Wars" as the query. Then the system returns fragments of XML data that are related to the query keywords as the result. Processing keyword search over static XML data has been extensively studied (Cohen *et al.*, 2003; Shao *et al.*, 2003; Sun *et al.*, 2007; Hristidis *et al.*, 2006; Liu *et al.*, 2007; Li *et al.*, 2013), but few research works addressed keyword search over XML streams (Vagena *et al.*, 2008; Hummel *et al.*; 2011; Gawinecki *et al.*, 2008).

Notice here that, in many applications, users are interested in limited fragments in XML data rather than entire data. For example, in the above example, one may only be interested in bibliographic information, such as title and author, but not in the contents of the books. In such cases, querying entire data does not make much sense. However, when using pure keyword search, we do not have such a control, that is, XML fragments that match with the query keywords will be included in the result even though they are in the part of XML data in which the user is not interested. For this reason, it is desirable if it is possible for us to be able to process keyword search for some restricted data.

Combining XPath with keyword search can solve the above problem. In fact, XQuery/XPath Full Text (Petkovic *et al.*, 2012) offers a solution. More precisely, XPath is used to specify the scope of keyword search in XML data, and query keywords are used to specify users' search demands. Combining these two query paradigms is beneficial to the users, because they can make use of the advantages from both query

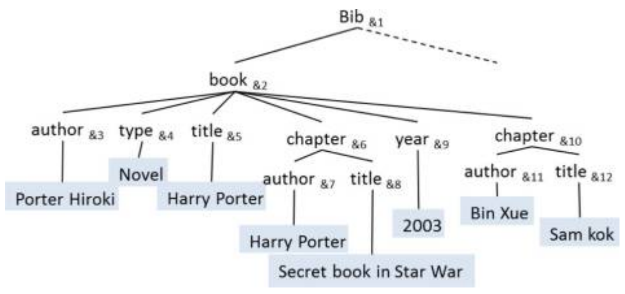


Figure 1.
An example of XML data

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

styles, that is, users do not need to fully understand the structure of the XML data being queried, while they can specify the area of interest in XML data in terms of XPath. For example, let us assume we are interested in book chapter that mention "Porter book". Such query can be represented as "//chapter[Porter book]". Such a combination can ensure that keywords "Porter" and "book" will be matched only in the subtrees rooted at element "chapter". Such XPath-based keyword search functionality has been successfully implemented in many systems that deal with static XML data; however, to the best of our knowledge, it has not been addressed in the context of XML streams, where XML data are continuously transmitted, although the popularity of XML streams has been growing rapidly.

To address this problem, we propose a scheme to process XPath-based keyword search over XML streams. More precisely, we try to extend nondeterministic finite automation (NFA)-based XPath filtering scheme for XML streams in such a way that it can deal with keyword-based filtering conditions. As for the NFA-based XML filtering engine, we exploit YFilter (Diao *et al.*, 2003), and we extend it by reference to CKStream (Hummel *et al.*, 2011), which is a pure keyword search algorithm for XML streams. In addition, as for the query result computation, we are based on meaningful lowest common ancestor (MLCA) semantics (Vagena *et al.*, 2007), whereas smallest lowest common ancestor (SLCA) semantics is used in CKStream, because MLCA returns more compact and meaningful query results than SLCA. We evaluate the proposed scheme by some experiments using both synthetic and real data sets. The experimental results show that our proposed method works well with acceptable throughputs, less memory usage, good efficiency and utility.

The rest of this paper is organized as follows. We review some preliminaries in Section 2 and some existing works in Section 3. We present the proposed algorithm and experimental evaluation in Sections 4 and 5, respectively. Finally, some related works are reviewed in Section 6, and Section 7 concludes this paper.

2. Preliminaries

First, we introduce some preliminaries.

2.1 XML path language (XPath)

XPath (Clark *et al.*, 1999) is a language that is used to search for XML fragments in a given XML data. We consider linear XPath expressions (Gupta *et al.*, 2004), *P*, given in (Gupta *et al.*, 2004) by the following grammar:

$$P{::} = /N \mid //N \mid PP$$
$$N{::} = E \mid A \mid tex() \mid text(S) \mid *$$

Here, *E*, *A* and *S* are an element label, an attribute label and a string constant, respectively, and * is the wild card. The function *text(S)* matches a text node whose value is the string *S*.

2.2 XML keyword search

XML Keyword Search is an XML search technique, where only keywords are used to formulate a query. The output is a set of ranked XML fragments matching the query

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

specification. The syntax of keyword search (Vagena *et al.*, 2008; Hummel *et al.*, 2011; Gawinecki *et al.*, 2008) is shown as follows.

2.2.1 Definition 1. An XML keyword search Q is a set of query keywords (t_1 […], t_m). Each query term is of the form: *l*::*k*, *l*::, ::*k* or *k*, where *l* is a node label and *k* is a keyword. A node n_i satisfies a query term of the form:

- *l*::*k* if n_i's label equals *l* and the tokenized text content of n_i contains the word *k*.
- *l*:: if n_i's label equals label *l*.
- ::*k* if the tokenized text content of n_i contains the word *k*.
- *k* if either n_i's label is *k* or the tokenized text content of n_i contains the word *k*.

To better illustrate the concept of keyword search, we present an example with a fragment of bibliography data source shown in Figure 1. From this data source, if a user wants to retrieve information on any publication which is written by author "Porter" and has type "Novel", he may issue a keyword search, q_1, with two keywords as "author::Porter type::Novel". Similarly, the user may issue a query, q_2, "author::Porter War" if he wants to know any publication which is about "War" and written by author "Porter". Note that, in these data, we observe that the word "book" appears in both XML element name and text value. Therefore, if the user issues keyword search, q_3, "book author::Hiroki", the keyword "book" matches with both XML element "book" whose ID is 2 and text value of XML element "title" whose ID is 8.

2.3 Node relatedness heuristics

In XML keyword search, given a set of query keywords, how to find XML fragments that are most eligible as the query results are quite important. Therefore, extensive studies have been done on these node relatedness heuristics (Cohen *et al.*, 2003; Gawinecki *et al.*, 2008; Hristidis *et al.*, 2006; Hummel *et al.*, 2011; Shao *et al.*, 2003; Vagena *et al.*, 2008; Xu *et al.*, 2005). Based on the fundamental method, lowest common ancestor (LCA), where the smallest XML fragments that subsume all keywords in the given keyword search are chosen as queries' results, many variants have been subsequently proposed. SLCA is one of the most popular one, which is defined as follows.

2.3.1 Definition 2. SLCA (Vagena *et al.*, 2007): two nodes n_1 and n_2 that belong to the same XML data d_i are meaningfully related if there are no nodes $n_3 \in d_i$ and $n_4 \in d_i$ such that LCA(n_3, n_4) is a descendant of LCA(n_1, n_2). If node $v \in d_i$, such that $v \in$ LCA(n_1, n_2), then we say that v is SLCA of n_1 and n_2, or $v \in$ SLCA(n_1, n_2).

However, SLCA semantics has a drawback that the results are too compact, and some correct search results tend to be discarded. For example, let us assume a keyword search "author::Porter title::", which is used to search for the title of any publication written by author "Porter". Based on SLCA semantics, the subtree rooted at node chapter ID 6 is returned; however, the subtree rooted at node book ID 2 is not returned as an answer, even though it is also the correct result of this query. MLCA semantics (Vagena *et al.*, 2007) has been proposed to address the above drawback. In this approach, the search result is selected in such a way that every two nodes are meaningfully related. MLCA is defined as follows.

2.3.2 Definition 3. MLCA (Vagena *et al.*, 2007): Let the set of nodes in an XML data be *N*. Two nodes are of the same type if and only if they have the same tag name. Given *A*,

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

$B \in N$, where A is composed of nodes of type α, and B is composed of nodes of type β, the MLCA set $C \subseteq N$ of A and B satisfies the following conditions:

- $\forall c_k \in C, \exists a_i \in A, b_j \in B$, such that $c_k = LCA(a_i, b_j)$. c_k is denoted as MLCA(a_i, b_j).
- $\forall a_i \in A, b_j \in B$, *if* $d_{ij} = LCA(a_i, b_j)$ and $d_{ij} \notin C$, then $\exists c_k \in C$, c_k is descendant of d_{ij}.
- Then set C is denoted as MLCASET(A, B).

According to MLCA, the answers of the above query are sets of all matched nodes in subtrees rooted at node Chapter ID 6 and book ID 2.

3. Existing works: YFilter and CKStream

3.1 YFilter

YFilter (Diao *et al.*, 2003) is a filtering system that processes XPath against XML streams. It provides real-time and fast matching of large numbers of XPath queries. The key innovation in YFilter is that it exploits prefix-sharing among all XPaths and generates a compact NFA. Therefore, the number of machine states in YFilter is small that can help speed up structure matching during the entire processing.

Figure 2 shows the NFA fragments of basic location steps. The state is represented by circle, and transition is represented by directed edge connecting two or more states. The symbol on an edge represents the incoming XML element that triggers the transition. The special symbol "*" matches any element. The symbol "ε" is used to denote an epsilon-transition. Figure 3 shows an example of such NFA corresponding to five XPath queries. A circle denotes a state. Circles with double lines denote accepting states, marked by the IDs of accepted queries.

YFilter relies on a SAX parser (Gorman *et al.*, 2004), which reads XML constructs, such as startElement and endElement, text content, etc. When startElement is received,

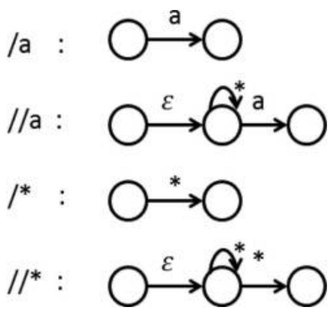


Figure 2.
Basic NFA location step

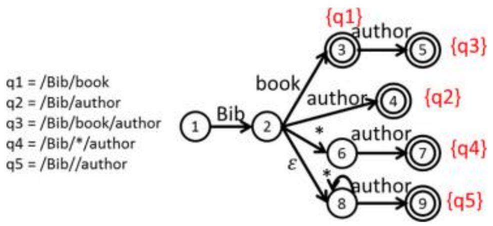


Figure 3.
XPath queries and a corresponding NFA

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

it triggers a state transition in the Finite State Machines (FSM), and must backtrack to previous states when endElement event is received. A run-time stack is used to track the active and previously processed states.

Given an XML data in Figure 1 and a set of queries in Figure 3, Figure 4 illustrates the run-time stack in the processing. When startElement event is invoked, an entry containing set of active states' IDs is pushed into the stack by following all matching transitions from the currently active states. First, if a transition marked by the incoming element name exists, the next state is added to the set of new active states. A transition marked by the "*" symbol is checked in the same way. Then, the state itself is added to the set. Finally, if an "ε"-transition exists, the state after the "ε"-transition is processed immediately according to these same rules.

3.2 CKStream

CKStream (Hummel *et al.*, 2011) is an algorithm to process multiple keyword queries over XML streams. It also relies on a SAX parser (Gorman *et al.*, 2004), parsing stacks and query indexes specially designed to allow the simultaneous matching of terms from different queries.

3.2.1 Parsing stack. Whenever XML constructs, such as startElement and endElement and text-content, are invoked, an entry is pushed into a stack, and that entry handles the following information: the label of the element; a bitmap, CAN_BE_SLCA, which contains one bit for each query; a set of used queries, which contains the IDs of the queries whose terms contain keywords; and keywords. Each entry is popped from the stack when the corresponding endElement event is invoked.

3.2.2 Query index. CKStream uses query index to deal with large number of queries efficiently. One unique query keyword is represented by an index entry, which also contains IDs of queries in which corresponding term occurs. Label and text value of query keywords are differentiated in query index. Figure 5 shows the query index of queries q_1, q_2 and q_3 above. Keywords on index 1, 2 and 4 are in the form *l*::*k*, whereas keywords on index 3 and 5 are in the form *k*. During processing, only one query index is used.

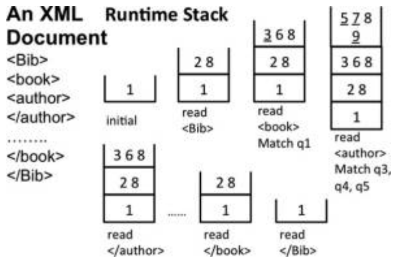


Figure 4.
An example of query processing in YFilter

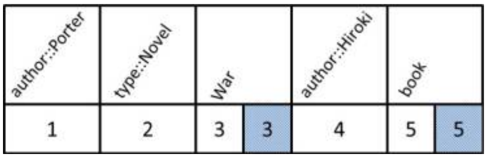


Figure 5.
A sample query index

3.2.3 Query bitmap. Each bit in query bitmap corresponds to one unique query keyword of its query's ID. One entry of the stacks is associated with one query bitmap representing all input keyword queries. Moreover, query bitmap is used to check and evaluate if the processed queries match or not. The information stored in query bitmap is similar to that of query index, but is used differently. There is only one query index in the filtering system, which is used to check if keywords are matched or not when processing the incoming SAX events. On the other hand, query bitmaps are more than one in the filtering system. The number of query bitmaps is the same as the number of entries pushed into the stack. Moreover, query bitmap is used to store information of the matched keywords, and is used to check and evaluate if the processed queries match or not. A query bitmap of queries q_1, q_2 and q_3 above is shown in Figure 6.

We explain how CKStream works as follows. When a startElement event is detected, it searches in the query index for the current node's label. If such node label exists in the query index, the position and query ID associated with such label will be obtained. Then the corresponding bit of the obtained position is set to true, and the query ID is inserted into the set of used queries. The CAN_BE_SLCA is set to true to all bits. Then the entry (label name, query bitmap, set of used query, CAN_BE_SLCA) is pushed into the stack. Similarly, when character() event is detected, the text value is split into words. For each single word and the combination of the label of its parent node with each word, it searches in the query index. If it exists in the query index, it sets the corresponding bits of the query bitmap and inserts the query ID to the set of used query of the entry in the stack of its parent node. Notice that terms of the form *l*::*k* are handled in the same way. Finally, when an endElement event is detected, it looks for complete queries. It first checks the set of used query to get all query IDs being processed. Then, for each processing query ID, it checks all the bits of the corresponding query ID in the query bitmap. If all bits of the corresponding query ID are not true, there is no query matched. Then it will update the entry of its parent node with its entry to be popped. Query bitmap will be concatenated, CAN_BE_SLCA will be combined using "AND" operator and set of used query will be combined together. Then the entry of the currently processed node will be popped out from the stack. If all bits of the corresponding query IDs are true, those specified queries are matched, and then the bit corresponding to the matched query is set to false in the query bitmap, and the matched query IDs will be removed from the set of used query. Then it will update the entry of its parent node with its entry as described above. The results will be returned to users or application, and the entry of the currently processed node will be popped out from the stack.

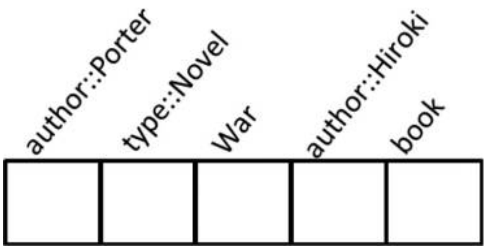


Figure 6. A sample query bitmap

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

4. Proposed method

4.1 Proposal overview

To make XPath-based keyword search possible, we attempt to integrate the method of NFA-based YFilter (Diao *et al.*, 2003) with CKStream [12] which is a state-of-the-art approach for keyword-based filtering over XML streams. To this end, we first extend the NFA-based finite state machine in YFilter so that all the four types of keywords are represented in term of NFA states. Next, we propose a method to generate the stack entry, which resulted from the integration of set of NFA states of YFilter with a set of used queries and query bitmap of CKStream, thereby combining the keyword-search capability of CKStream and the path-based filtering of YFilter. In addition, we use MLCA heuristic to generate query results, whereas SLCA is used in CKStream, because it gives more related and compact results than other heuristics (Vagena *et al.*, 2007). MLCA returns a set of nodes that match queries.

4.2 Proposed algorithm

4.2.1 Query syntax. To allow XPath-based keyword search, we use keywords to specify a query predicate in an XPath expression. In fact, XQuery and XPath Full Text 1.0 (Petkovic *et al.*, 2012) is a W3C standard for that purpose, but its full syntax is too complicated. For this reason, we borrow the core syntax from it. The resulted syntax is as follows:

$$/XPath[ftcontains(keyword - search\ query)]$$

where *XPath* is an XPath expression and *ftcontains* is a dedicated function to specify a keyword search according to (Vagena *et al.*, 2008; Hummel *et al.*, 2011; Gawinecki *et al.*, 2008). Note that the XPath and keyword search are connected by a descendant axis. Note also that one might argue that writing XPath still requires users to know well about the structure of the XML stream being queried. In fact, we leave the freedom for users to specify any XPath expressions, that is, an extreme case is to use simple expressions of the form "//element" if one only knows the element name of interest, while more complicated XPath can be used if one understands the detailed structure of the XML stream.

For example, if a user would like to search for "chapter" which contains the words "Porter" and "book", he can easily combine an XPath "//chapter" with keywords "Porter" and "book", which results in "//chapter[ftcontains(Porter book)]". Such combination can ensure that keywords "Porter" and "book" will be searched only inside the subtrees rooted at element "chapter".

4.2.2 Extension of NFA in YFilter. In the next, we extend NFA in YFilter so that we can encode user-specified query keywords in terms of states in NFA, thereby making it possible to process keyword search over XML streams using NFA-based Finite State Machine. More precisely, in the original NFA model, a state transition happens only when an XML element is parsed. We modify the NFA model as follows:

- As mentioned earlier, there are four types of query keywords, namely, l::k, l::, ::k or k. We introduce new state transitions corresponding to these types of query keywords where the transitions start with epsilon-transition (ε) and *-transition in this order (Figure 7);

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

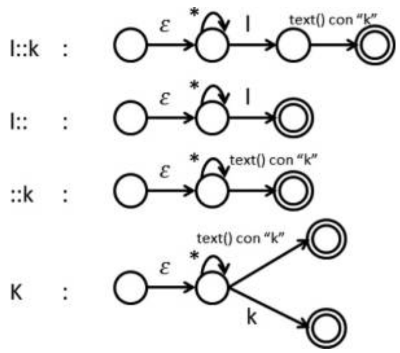


Figure 7. Basic extended-NFA location step

- To deal with text contents, we introduce new state transition denoted by "text()", which is to make transition when a text content event is detected; and
- Each accepting state in the extended NFA contains the position of bit inside the query bitmap and the matched query IDs. To check if a query is matched, we check all corresponding bits in the query bitmap.

4.2.3 Details of algorithm. Having extended the NFA model according to the four types of query keywords, we can now represent an XPath-based keyword query in terms of an NFA. Next, given such a query, we explain how it is processed against an XML stream. Specifically, we elaborate how to operate a running stack during the process.

Each entry in the stack contains three pieces of information: the set of NFA states, the set of used queries and the query_bitmap. The entry is pushed or popped from the stack when detecting startElement or endElement event, respectively. Each used query from the set used_query is evaluated by checking if all corresponding bits in query bitmap are true.

The details of our proposed algorithm are shown in Algorithm 1. There are three main functions: Callback Function Start of Element, Callback Function Text and Callback Function End of Element.

(1) *Callback function start of element*: This function is invoked when the start Element is called. An empty set of used queries and a query bitmap, initialized to false by default, are created. Then, starting from the initial or current states, state transitions are processed according to startElement. For those states that are newly visited, the query IDs and the positions of bits will be obtained, and they are put in the set of used queries and corresponding bits are set to true in the query bitmap, if any of those states are accepting states. If no accepting state exists, all bits in the query bitmap remain false and the set of used queries is empty. Finally, the set of active states, query bitmap and set of used queries are inserted into an stack entry, and are pushed into the stack.

(2) *Callback function text*: When the character() event that corresponds to text content is called, the text is split into tokens, and each token is processed by NFA. Following the same procedure as mentioned in Callback Function Start of Element, the information that is obtained from the accepting states is used to update the entry at the top of the stack (the entry of parent nodes in the stack). In

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

Callback Function Text, neither new query bitmap nor new set of used queries is created. As a consequence, no entry is pushed into the stack.

(3) *Callback function end of element*: When an endElement event is called, the top entry is popped out from the stack. Then all query IDs in the set of used queries in the popped entry are checked. For each query ID being processed, all bits in the query bitmap of the corresponding query are checked if they are all true. If so, the corresponding query is matched, and the results are returned; otherwise, if all bits of the corresponding query are true, then those corresponding query IDs are added to the set of used queries in the top entry of the stack, and all bits in the query bitmap of the popped entry are used to update the query bitmap in the top entry by applying bit-wise OR operation.

```
Algorithm 1 The Proposed Method Callback Functions
Callback Function Start of Element
Input:     Parsing stack S, the XML node e being processed
 1: Initialization
 2: Push(sn) {create new stack entry}
 3: Process e against extended-NFA
 4: Add the newly active states to the stack
 5: N := number of distinct terms in all queries being processed
 6: sn.query_bitmap [0, …, N-1]: = false
 7: while each newly active state do
 8:     if sn.state is accepting state then
 9:         p := get position of query bitmap
10:         q := get query ID of keyword matched
11:         Add q to sn.used_queries
12:         sn.query_bitmap[p]: = true
13:     end if
14: end while
Callback Function Text
Input:     Parsing stack S, the XML node e being processed
 1: initialization
 2: sn := *top(S) {sn points to the top entry in the stack}
 3: K := set of tokens in node e
 4: while all k ∈ K do
 5:     Process k against extended-NFA
 6:     Add the newly active states to the stack
 7:     while each newly active state do
 8:         if sn.state is accepting state then
 9:             p := get position of query bitmap of term l::k
10:             q := get query ID of keyword matched
11:             Add q to sn.used_queries
12:             sn.query_bitmap[p]: = true
13:         end if
14:     end while
15: end while
Callback Function End of Element
Input:     Parsing stack S, the XML node e being processed
 1: Initialization
 2: sn := pop(S) {pops the top entry in the stack to sn}
```

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

```
3:  tn := *top(S) tn points to the top entry in the stack
4:  while q ∈ sn.used_queries do
5:     let $j_1$ …, $j_N$ be the positions of the bits corresponding to terms from query q in
       query_bitmap
6:     COMPLETE := sn.query_bitmap [$j_1$] and … and sn.query_bitmap [$j_N$]
7:     if sn.state is accepting state then
8:        q.results := q.results ∪ sn
9:     else
10:       Add sn.used_queries to tn.used_queries
11:       tn.query_bitmap := sn.query_bitmap or tn.query_bitmap
12:    end if
13: end while
```

We show how our proposed method works using an example with two keyword queries as follows. They are also illustrated in Figures 8 and 9:

- *Q1*: //book[ftcontains(author::Porter type::Novel)]
- *Q2*: /bib/book/chapter[ftcontains(author::Porter War)]

As shown in Figure 9, the initial state is initialized when new XML data, startDocument, is detected. When receiving an event startElement or character event, the system pushes an entry into the stack. When receiving an event endElement, the system checks for matched queries. If no query matches, all bitmaps of the query_bitmap are sent and combined with the entry at the stack top using bit-wise OR operation. The set used_query of the popped entry is added up to that of the top entry. Then, the entry is popped out of the stack.

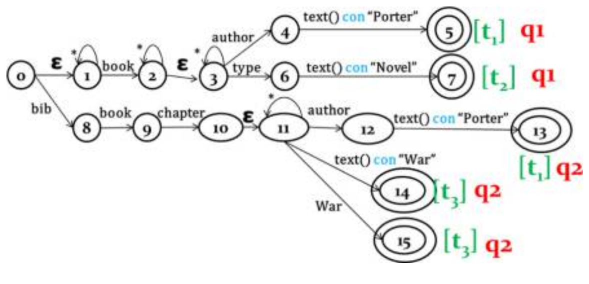


Figure 8. A single extended-NFA

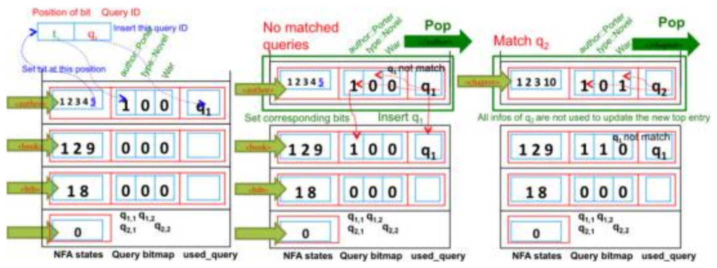


Figure 9. A running example of the proposed method

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

5. Experimental evaluation

5.1 Experimental overview

Our experiments are done as described below. Experimental setup is explained in Section 5.1. We compare the performance of our proposed method by using XPath-based keyword search with that of CKStream by using keyword search on queries with same search intentions in Section 5.2.1. In Section 5.2.2, we explain how well our proposed method can deal with the increase of number of queries and keywords by using XPath-based keyword search. Finally, we compare the throughputs of our proposed method with that of YFilter by using XPath queries in Section 5.2.3.

5.2 Experimental setup

We implemented our proposed method in Java based on the existing YFilter (Diao *et al.*, 2003) and the SAX API from Xerces Java Parser (Gorman *et al.*, 2004). All data structures, query bitmaps and sets of used queries are stored as in-memory data structures. All experiments are performed in an Intel Core 2.33 GHz PC with 2 GB memory running Windows XP Service Pack 2.

We use two types of data sets: synthetic and real data. The synthetic data are generated by the xmlgen of XMark (Busse *et al.*, 2013). The real data are DBLP bibliography (CSE *et al.*, 2013a) and Mondial (CSE *et al.*, 2013b) world geographic database. The details of the data sets are presented in Table I.

5.3 Experimental results

5.3.1 Performance comparison on queries with same search intentions. First experiment is to evaluate the accuracy of our proposed method. Specifically, we compare the proposed scheme with CKStream (Hummel *et al.*, 2011), because it is based on SLCA semantics (Vagena *et al.*, 2007), whereas our scheme is based on MLCA semantics. The queries used are listed in Tables II and III. They are chosen randomly from the XML data generated by XMark. The relevant matches of the search intentions in Table II are the subtrees rooted at “auction”, “shipping”, “category”, “people” and “regions” that contain the query keywords. Note that, for experimental purpose, we modify XML data set by adding several keywords into randomly chosen text nodes. The translated

Table I.
All data sets used in the experiment

Data set	Maximum depth	Average depth	Size (MB)
XMark (Busse *et al.*, 2013)	5	3	11.7
DBLP (CSE *et al.*, 2013a)	6	2.90	15.7
Mondial (CSE *et al.*, 2013b)	5	3.59	1.8

Table II.
Search intentions and all translated keyword searches

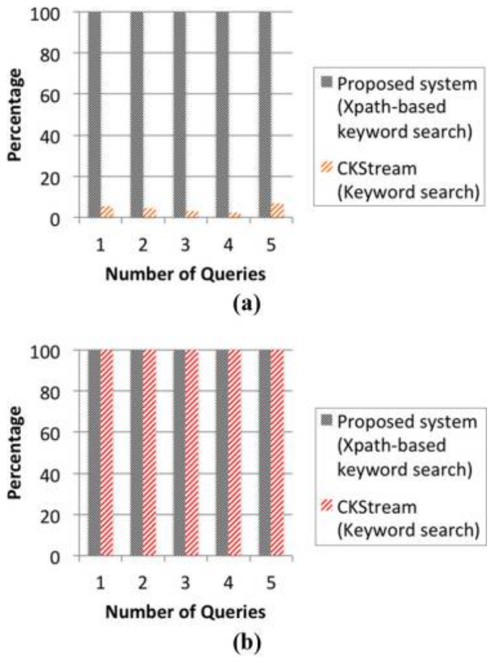
No.	Search intentions	Keyword search
1	Find the “auction” that is related with “milk” and “toothpaste”	Auction milk toothpaste
2	Find the “shipping” with “fixed” “pays”	Shipping fixed pays
3	Find the “category” that is related with “grape” and “roses”	Category grape roses
4	Find the “people” who belongs to “Democratic” and “Republic”	People democratic republic
5	Find the “regions” that “payment” is done by “cash”	Regions payment cash

keyword searches in Table II are processed in CKStream, and the translated XPath-based keyword searches in Table III are processed in our proposed method.

5.3.1.1 Accuracy. Next, we evaluate the effectiveness of our proposed method against CKStream based on precision, recall and F-measure. F-measure is the weighted harmonic mean of precision and recall. As shown in Figure 10(a), CKStream has low precision on all queries, because it returns many unrelated results (any subtrees that contain all keywords), whereas our proposed method has high precision (100 per cent), because it only returns relevant matches. This suggests the effectiveness of the XPath-based filtering in keyword search, which can help users to filter out unnecessary parts from the results. Regarding recall, both methods have high recall (100 per cent) as shown in Figure 10(b). In addition, we calculated the average F-measure as shown in Table IV of the queries shown in Tables II and III. We can observe that our proposed

Table III. The translated XPath-based keyword searches from the search intentions shown in Table II

No.	XPath-based keyword search
1	//auction[ftcontains(milk toothpaste)]
2	//shipping[ftcontains(fixed pays)]
3	//category[ftcontains(grape roses)]
4	//people[ftcontains(Democratic Republic)]
5	//regions[ftcontains(payment cash)]



Notes: (a) Precision; (b) recall

Figure 10. Precision and recall

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

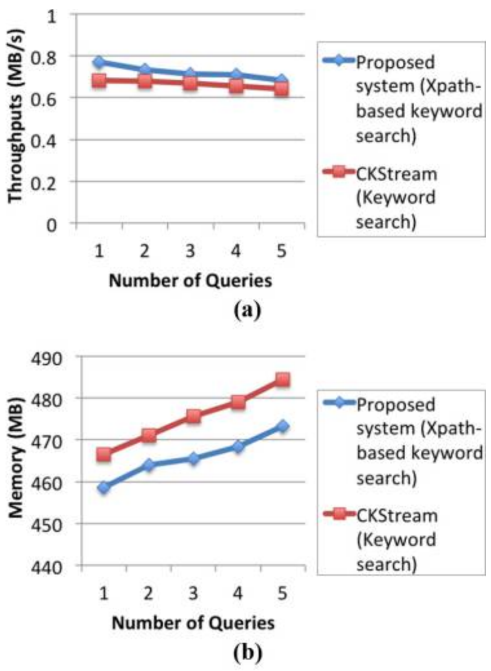
method achieves much higher *F*-measure than CKStream, because CKStream returns many undesired results as explained earlier.

5.3.1.2 Throughputs and memory usages. Next, we evaluate the performance of our proposed method and CKStream based on their throughputs and memory usage when querying for the search intentions in Table II. As shown in Figures 11(a and b), our proposed method produces higher throughputs and consumes less memory than CKStream because, in our proposed method, the respective keywords shown in Table III are searched only inside the subtrees rooted at elements "auction", "shipping", "category", "people" and "regions", whereas CKStream tries to search for all keywords shown in Table II in the entire XML data. Such unnecessary searching causes the querying performance worse. As a result, our proposed method enjoys producing higher throughputs and using less memory consumption.

5.3.2 Scalability. Next experiment is to measure the scalability of our proposed method when the number of XPath-based keyword search is increased. We measure the throughputs for processing all XML data in Table I. Similarly, we measured the average memory usage and number of extended-NFA states while processing each XML data.

Table IV. Comparison on *F*-Measure

F-Measure	CKStream	Proposed scheme
XMark	0.126	1



Notes: (a) Throughputs; (b) memory usages

Figure 11. Proposed system vs. CKStream on queries with the same search intentions

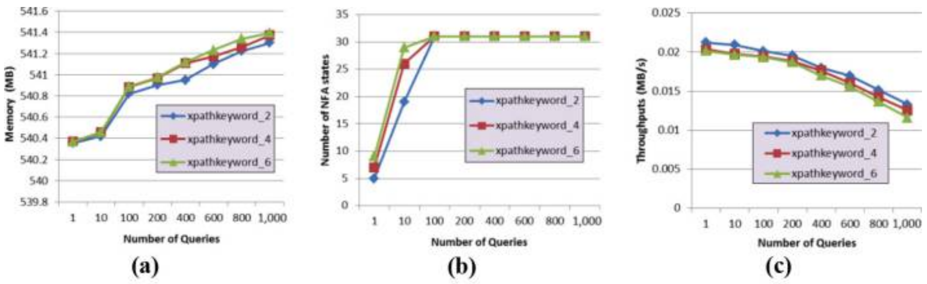
Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

We generate XPath-based keyword search from the above data sets. Since there is a performance impact when searching for different kinds of keywords in our system, we separately generate sets of queries which contain only keywords in the forms of *l*::, *l*, *l*::*k*, ::*k* and mixture of the four forms. For example, we could generate: //regions[ftcontains(:: begin ::caius)], whose keywords are in the form of ::*k* from XMark data set. Moreover, since the same keywords can appear in different queries, we divide our sets of queries into two categories, sets of queries in which the same keywords can appear in different queries and sets of queries in which the same keywords cannot appear in different queries. For the XPath-part, the depth is 2.

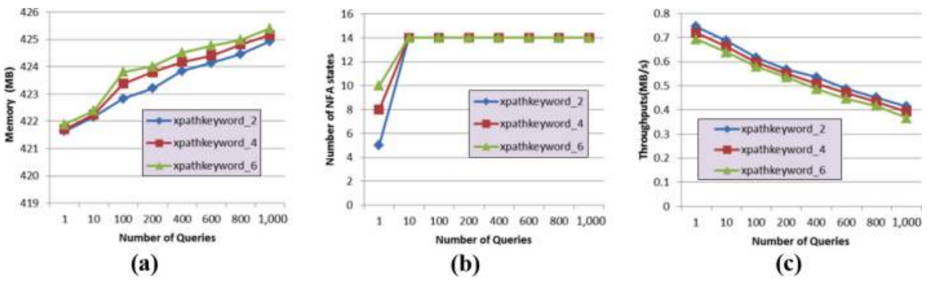
5.3.2.1 Varying the number of queries. We investigate the impact on the performance of the algorithm when the number of queries increases. We vary the number of queries from 1, 10, 100, 200, 400, 600, 800 and 1,000. We observe that as the number of queries increases, the memory usage increases, and the number of NFA states also increases while the throughputs constantly decrease as shown in Figures 12-23. This is because, the system needs to process each single query and output the result of each queries.

5.3.2.2 Varying the number of query keywords. We investigate the impact on the performance of our algorithm when we increase the number of keywords from two, four and six. With the sets of queries whose same keywords can appear in more than one query, we observed that though the number of keywords increase, the memory usage, number of extended-NFA states and throughputs do not change much between two, four and six keywords. These cause by the more frequency that same keywords appear at different queries as shown in Figures 12-14.



Notes: (a) Memory usage; (b) NFA states; (c) throughputs

Figure 12. DBLP: varying the number of queries and keywords of type *l*::



Notes: (a) Memory usage; (b) NFA states; (c) throughputs

Figure 13. Mondial: varying the number of queries and keywords of type *l*::

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

Next we investigate the impact that the increase in unique keywords in the same set of queries has on the performance of the algorithm. As expected, as the number of unique keywords increases, the number of NFA states also increases. As a result, the memory usage increases and the throughputs decrease significantly as shown in Figures 15-23. Though memory usage and throughputs of the algorithm have some degradation when the number of queries and the number of keywords increase, the algorithm scales well with such increase.

5.3.3 Performance comparison on pure keyword search and XPath. Next, we investigate the overheads that our proposed work can deal with pure XPath and pure keyword search. Instead of using the queries shown in Tables II and III. We generate

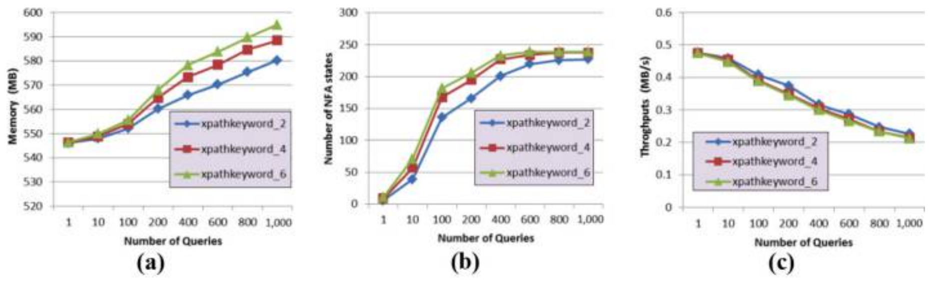


Figure 14. XMark: varying the number of queries and keywords of type *l*::

Notes: (a) Memory usage; (b) NFA states; (c) throughputs

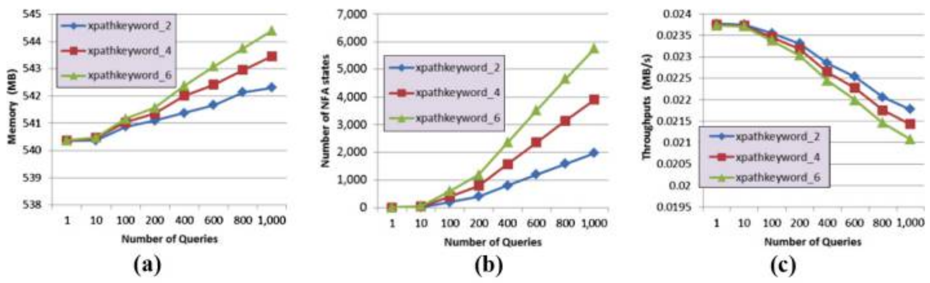


Figure 15. DBLP: varying the number of queries and keywords of type ::*k*

Notes: (a) Memory usage; (b) NFA states; (c) throughputs

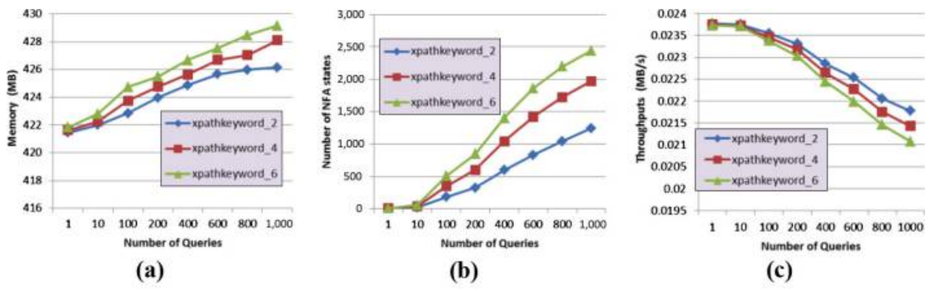


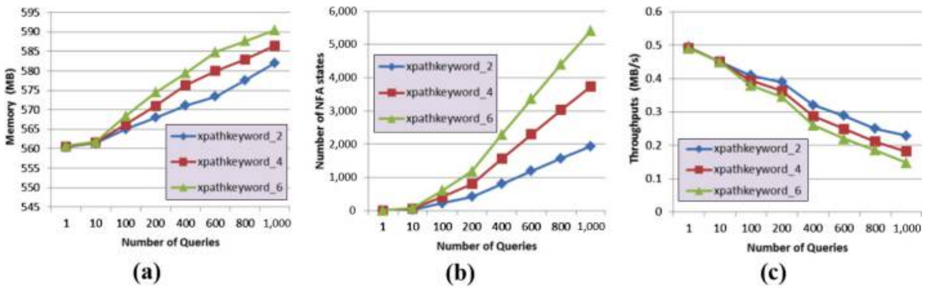
Figure 16. Mondial: varying the number of queries and keywords of type ::*k*

Notes: (a) Memory usage; (b) NFA states; (c) throughputs

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

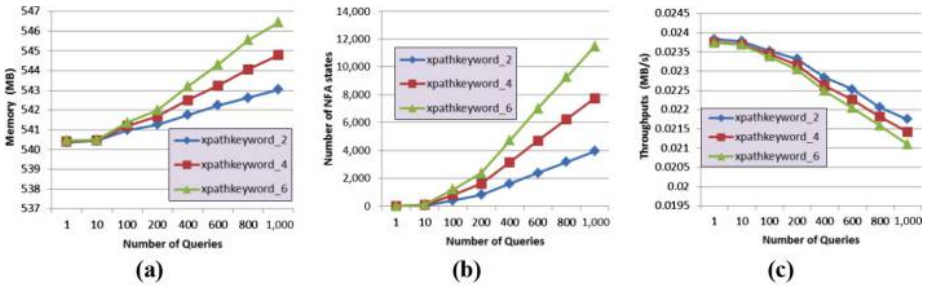
pure XPath and pure keyword search from the synthetic data set XMark (Busse *et al.*, 2013). For XPath, the minimum depth is 3 and maximum depth is 5. There is no "*" and "//" axis in the XPath. For keyword search, we used the set of queries with two, four and six keywords, and the types of keywords were randomly generated in the form of "*l*::", "*k*", "*l*::*k*" and "::*k*". We then calculated the average throughputs of the set of queries with two, four and six keywords.

First, we compare the performance of CKStream (Hummel *et al.*, 2011) with our proposed work by investigating on their throughputs. As shown in Figure 24, our proposed work performs worse than CKStream (Hummel *et al.*, 2011) when processing



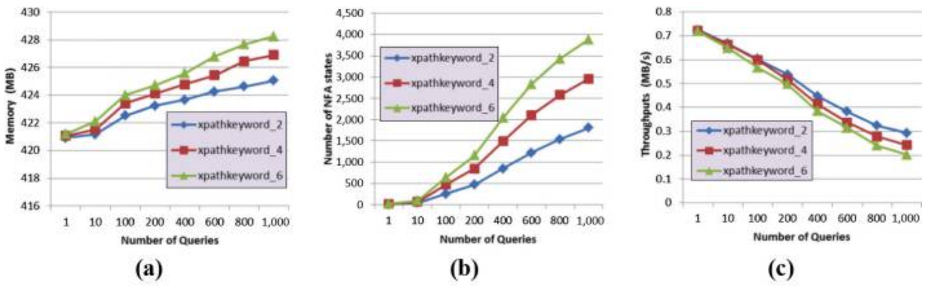
Notes: (a) Memory usage; (b) NFA states; (c) throughputs

Figure 17. XMark: varying the number of queries and keywords of type ::*k*



Notes: (a) Memory usage; (b) NFA states; (c) throughputs

Figure 18. DBLP: varying the number of queries and keywords of type *k*



Notes: (a) Memory usage; (b) NFA states; (c) throughputs

Figure 19. Mondial: varying the number of queries and keywords of type *k*

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

pure keyword search. This is caused by the size of our NFA. As explained earlier, when the keyword type is in the form of "*k*", which is matched to either XML element name or text content, the number of NFA states are two. As a consequence, the size of NFA is bigger than the size of query index of CKStream. This causes the time to do the index look up in the query index less than the time to traverse from states to states in NFA of our proposed method. Though, CKStream performs better than our proposed method when processing pure keyword search at only around 7.41 per cent.

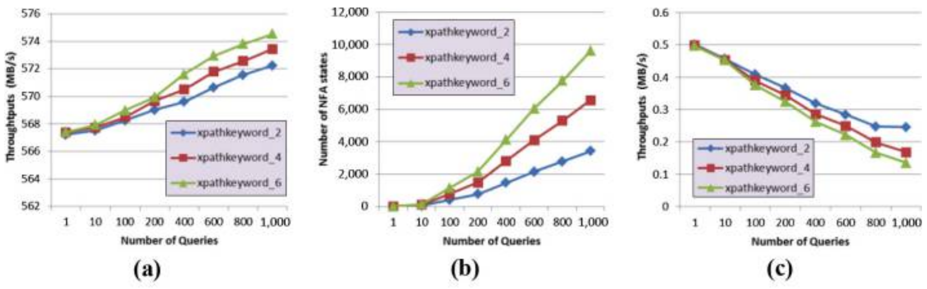


Figure 20.
XMark: varying the number of queries and keywords of type *k*

Notes: (a) Memory usage; (b) NFA states; (c) throughputs

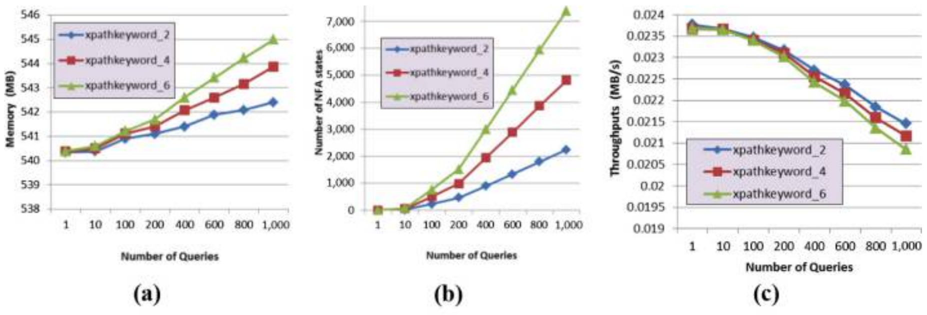


Figure 21.
DBLP: varying the number of queries and keywords of mixed types *l*::, ::*k*, *l*::*k* and *k*

Notes: (a) Memory usage; (b) NFA states; (c) throughputs

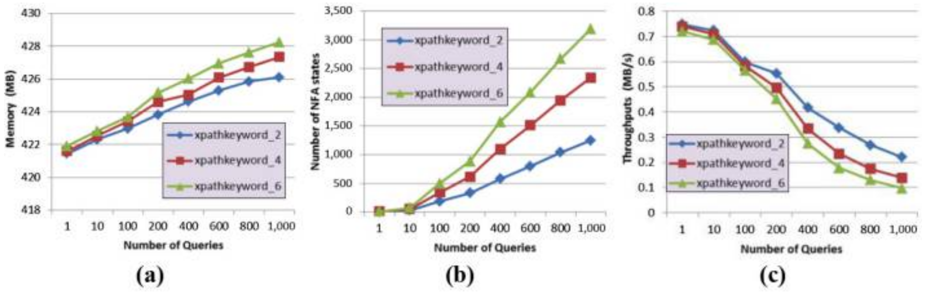
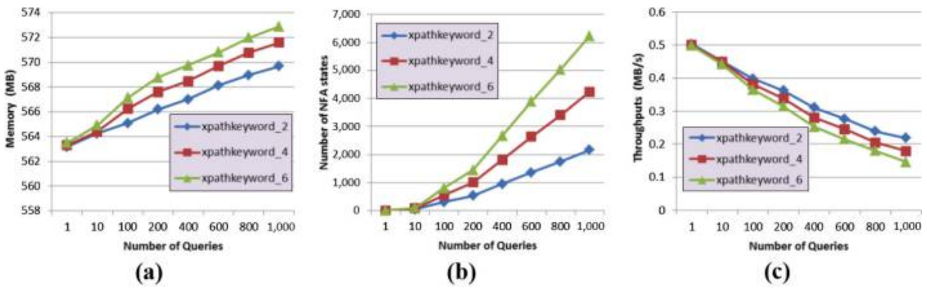


Figure 22.
Mondial: varying the number of queries and keywords of mixed types, *l*::, ::*k*, *l*::*k* and *k*

Notes: (a) Memory usage; (b) NFA states; (c) throughputs

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

Next, we compare the throughputs of YFilter (Diao *et al.*, 2003) with our proposed work as shown in Figure 25. As expected, YFilter performs much better. This is caused by the fact that the entry of the stack in YFilter is associated with only the set of NFA states, whereas in our proposed work, additional running time is needed to create and update the query bitmap and the set of used query. Moreover, in YFilter, when the accepting states are reached, the processed queries are evaluated to be matched and return the query results at the mean time; however, in our proposed work, in addition to checking for accepting states in the entry, we need to get information from the accepting states and update the query bitmap and the set of used queries. These tasks cause much overhead to our proposed work. Though our proposed work is able to provide less throughputs than the existing works, CKStream and YFilter, when processing pure keyword search and XPath, it is more flexible and able to deal with more varieties of query types. YFilter can only process XPath, and CKStream can only handle keyword



Notes: (a) Memory usage; (b) NFA states; (c) throughputs

Figure 23. XMark: varying the number of queries and keywords of mixed types, *l*::, ::*k*, *l*::*k* and *k*

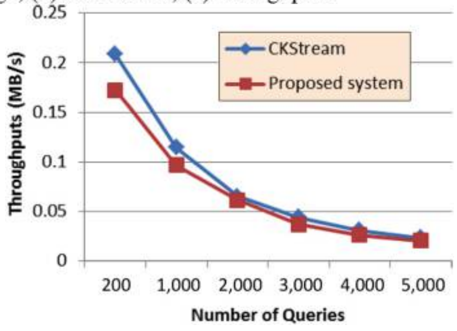


Figure 24. Throughputs of proposed system and CKStream

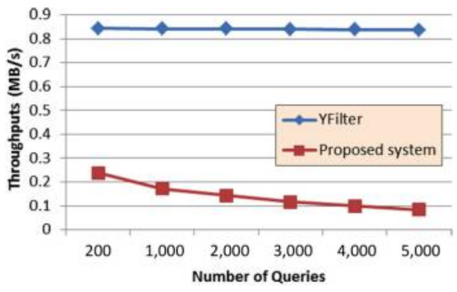


Figure 25. Throughputs of proposed system and YFilter

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

search; however, our proposed work can handle those two types of queries plus the more innovative query type, XPath-based keyword search. This makes our proposed work more realistic in real world scenario.

6. Related works

In this section, we explain some related works that made significant contributions to keyword search and XPath over XML and XML streams.

6.1 Keyword-based search

Using keywords to query XML databases has been extensively studied. XSEarch (Cohen *et al.*, 2003) adopts an intuitive concept of meaningfully related sets of nodes based on the relationship of the nodes with its ancestors (on the graph). Its query language is simple and suitable for a naive user. Moreover, XSEarch deals efficiently with large documents because it adopted index structures and evaluation algorithms (Cohen *et al.*, 2003).

XRANK (Shao *et al.*, 2003) presents an adaptation of Google's PageRank to XML data for computing ranking scores for the answer trees. It computes the rank of XML elements, which takes into account both hyperlinks and containment edges. For containment edges, it considers a two-dimensional proximity metric involving both the keyword distance and ancestor distance.

The works in (Vagena *et al.*, 2008; Hummel *et al.*, 2011; Gawinecki *et al.*, 2008) took the first step towards processing keyword search over XML streams. In (Vagena *et al.*, 2008), they introduced sophisticated query processing algorithms that can answer keyword search over streaming XML data. This work is more novel than the previous works, which mainly worked on static XML data; however, this work only supports single keyword search over XML streams. To fulfill this incompleteness, the work in (Hummel *et al.*, 2011) proposed multiple keyword searches over XML streams. They proposed two new algorithms, KStream and CKStream, for simultaneously processing several keyword searches over XML streams.

6.2 XPath-based search

Recent research on XML streams (dissemination, filtering and routing) (Campillo *et al.*, 2003; Han *et al.*, 2005; Onizuka *et al.*, 2003; Lu *et al.*, 2006; Zhou *et al.*, 2006; Yu *et al.*, 2004) aims at building large-scale, distributed systems (Gupta *et al.*, 2004; Diao *et al.*, 2003; Jaehoon *et al.*, 2007; Candan *et al.*, 2006). In (Diao *et al.*, 2003), they developed YFilter, an XML filtering system aiming at providing efficient filtering for large numbers of XPath queries. The commonality among XPath queries are well-studied and they found out that, by merging the common prefixes of the XPaths, the machine states become very compact. To deal with multiple XML schemas, two approaches are very common: to apply query rewriting (Yu *et al.*, 2004) and to use global schemas (Li *et al.*, 2004). Gupta *et al.* (2004) proposed a lazy deterministic finite automation (DFA)-based filtering system, which is superior to the NFA-based filtering system in terms of processing performance; however, there are several drawbacks of the DFA-based filtering system, one of which is its excessive consumption of memory caused by a large number of DFA states, and thus, the system can run out of the memory.

PosFilter was proposed in (Jaehoon *et al.*, 2007) to address the problem of XML filterings that exploit the prefix commonalities among path expressions. Such prefix-path-sharing systems suffer from the explosion of NFA states when XPath

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

expressions contain ancestor-descendant axis ("//"). If XPath expressions that begin with ancestor-descendant axis ("//") are used often, such queries are most likely to have the postfix sharing. Such NFA state explosion can be solved by exploiting postfix sharing among XPath expressions.

AFilter (Candan *et al.*, 2006) was proposed to take advantage of both prefix and postfix sharing to reduce the overall filtering time and increase throughputs. Different from previous works, AFilter makes use of the capability of both prefix and postfix commonalities among XPath expressions. AFilter provides balance between memory usage and performance speed up.

7. Conclusion

In this paper, we have developed a filtering system that supports XPath, keyword search and XPath-based keyword search over XML streams. The NFA model is extended so that it supports XPath-based keyword search query. We have also integrated the method used in YFilter with that of CKStream by using the above extended-NFA so that it supports the above query types.

We evaluate them by some experiments on both synthetic and real data sets. The experimental results show that our proposed method works well with acceptable throughputs, less memory usage and good efficiency and utility.

For our future work, we are going to apply our proposed method to more complex topics such as multimedia keyword filtering and to multiple XML streams, and useful information can be obtained when information from different sources is combined at real-time.

References

Bray, T., Paoli, J., Sperberge-McQueen, C.M., Maler, E. and Yergeau, F. (2008), "Extensible markup language (XML) 1.0 (fifth edition) – world wide web consortium proposed recommendation", available at: www.w3.org/TR/xml

Busse, R., Carey, M., Florescu, D., Kersten, M., Manolescu, I., Schimdt, A. and Wass, F. (2013), "XMark: an XML benchmark project", available at: www.xml-benchmark.org/

Campillo, I., Green, T., Suciu, A. and Onizuka, M. (2003), *XMLTK: An XML Toolkit for Scalable XML Stream Processing*, University of Washington, Seattle, WA.

Candan, K.S., Hsiung, W., Chen, S., Tatemura, J. and Agrawal, D. (2006), "A filter: adaptable XML filtering with prefix-caching and suffix clustering", *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul*, pp. 559-570.

Clark, J. and DeRose, S. (1999), "XML path language (XPath) version 1.0 – world wide web consortium recommendation", available at: www.w3.org/TR/xpath

Cohen, S., Mamou, J., Kanza, Y. and Sagiv, Y. (2003), "XSEarch: a semantic search engine for XML", *Proceedings of the 29th International Conference on Very Large Data Bases, Berlin*, pp. 45-56.

CSE, U., Database, U. and Suciu, D. (2013a), "Dblp dataset: computer science bibliography", available at: www/repository.html/dblp

CSE, U., Database, U. and Suciu, D. (2013b), "Mondial dataset: world geographic database", available at: www/repository.html/mondial

Diao, Y. and Franklin, M.J. (2003), "High-performance XML filtering: an overview of YFilter", *IEEE Data Engineering Bulletin*, Vol. 26 No. 1, pp. 41-48.

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)

Gawinecki, M., Mandreoli, F. and Cabri, G. (2008), *Keyword Search Over XML Streams: Addressing Timestamping and Understanding Results*, University of Modena, Modena, pp. 371-382.

Gorman, E. (2004), "Generic XML stream parser API: an easier way to use SAX and Xerces", available at: http://xml.apache.org/xerces-j/index.html

Gupta, A., Green, J. and Onozuka, M. (2004), "Processing XML stream with deterministic automata and stream indexes", *ACM Transactions on Database Systems (TODS*), Vol. 29 No. 4, pp. 752-788.

Han, J., Chen, Y., Dong, G., Pei, J., Wah, B., Wang, J. and Cai, Y. (2005), "Stream cube: an architecture for multidimentional analysis of data streams", *Distributed and Parallel Databases*, Vol. 18 No. 2, pp. 173-197.

Hristidis, V., Koudas, N., Papakonstantinou, Y. and Srivastava, D. (2006), "Keyword proximity search in XML trees", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 18 No. 4, pp. 525-539.

Hummel, C., da Silva, S., Moro, M. and Laender, H.F. (2011), "Multiple keyword-based queries over XML streams", *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, *New York, NY*, pp. 1577-1582.

Jaehoon, K., Youngsoo, K. and Seog, P. (2007), "PosFilter: an efficient filtering technique of XML documents based on postfix sharing", *24th British National Conference on Databases, Glasgow*, pp. 70-81.

Li, J., Wang, J. and Huang, M. (2013), "Exploiting the relationship between keywords for efficient XML keyword search", *Advances in Databases and Information Systems-17th East European Conference, ADBIS 2013* , *Genoa*, *1-4 September*, pp. 232-245.

Li, Y., Yu, C. and Jagadish, H.V. (2004), "Schema-free XQuery", *Proceedings of the 13th International Conference on Very Large Databases*, *San Francisco, CA*, pp. 72-83.

Liu, Z. and Chen, Y. (2007), "Identifying meaningful return information for XML keyword search", *Proceeding of the 2007 ACM SIGMOD International Conference on Management Data*, *Beijing*, pp. 329-340.

Lu, W., Chiu, K. and Pan, Y. (2006), "A parallel approach to XML parsing", *Proceeding of the 7th IEEE/ACM International Conference on Grid Computing*, *Barcelona*, *28-29 September 2006*, pp. 223-230.

Onizuka, M. (2003), "Light-weight XPath processing of XML stream with deterministic automata", *Proceeding of the 12th International Conference on Information and Knowledge Management*, *Los Angeles, CA*, pp. 342-349.

Petkovic, D. (2012), "XPath and XQuery full text standard and its support in RDBMSs", available at: www.w3.org/TR/xpath-full-text-10/

Shao, F., Guo, L., Botev, C. and Shanmugasundaram, J. (2003), "XRANK: ranked keyword search over XML documents", *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, *New York, NY*, pp. 16-27.

Sun, C., Chan, C. and Goenka, A. (2007), "Multiway SLCA-based keyword search in XML data", *Proceedings of the 16th International Conference on World Wide Web*, *Alberta*, pp. 1043-1052.

Vagena, Z., Colby, L.S., Ozcan, F., Balmin, A. and Li, Q. (2007), "On the effectiveness of flexible querying heuristics for XML data", *Proceedings of 5th International XML Database Symposium*, *Vienna*, pp. 77-91.

Vagena, Z. and Moro, M. (2008), "Semantic search over XML document streams", *Proceedings of 3rd International Workshop on Database Technologies for Handling XML Information on the Web, Nantes*.

Xu, Y. and Papakonstantinou, Y. (2005), "Efficient keyword search for smallest LCAs in XML databases", *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, New York, NY*, pp. 527-538.

Yu, C. and Popa, L. (2004), "Constraint-based XML query rewriting for data integration", *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, New York, NY*, pp. 371-382.

Zhou, X., Thakkar, H. and Zaniolo, C. (2006), "Unifying the processing of XML streams and relational data streams", *Proceeding of the 22nd International Conference on Data Engineering, Los Angeles, CA*, p. 50.

Further reading:

Bou, S., Amagasa, T. and Kitagawa, H. (2014), "Filtering XML streams by XPath and keywords", *Proceeding of the 16th International Conference on Information Integration and Web-Based Applications & Services, Hanoi*, pp. 410-419.

About the authors

Savong Bou received B.E. degree in Computer Science from Norton University in 2009. In 2014, he received his M.E. degree from the Department of Computer Science, University of Tsukuba. He is currently a Ph.D. student at the Graduate School of Systems and Information Engineering, University of Tsukuba. His research interests include databases, data mining, stream processing and information retrieval. Savong Bou is the corresponding author and can be contacted at: savong.bou@kde.cs.tsukuba.ac.jp

Toshiyuki Amagasa is an associate professor at Faculty of Engineering, Information and Systems, University of Tsukuba. His research interests include data engineering, web information management, scientific information management and data mining. He is a senior member of IEICE and IEEE, and a member of DBSJ, IPSJ and ACM.

Hiroyuki Kitagawa received the B.Sc. degree in physics and the M.Sc. and Dr.Sc. degrees in computer science, all from the University of Tokyo, in 1978, 1980 and 1987, respectively. He is currently a full professor at Faculty of Engineering, Information and Systems and at Center for Computational Sciences, University of Tsukuba. He serves as President of the Database Society of Japan. His research interests include data integration, databases, data mining and information retrieval. He is an IEICE Fellow, an IPSJ Fellow, a member of ACM and IEEE and an Associate Member of the Science Council of Japan.

For instructions on how to order reprints of this article, please visit our website:
www.emeraldgrouppublishing.com/licensing/reprints.htm
Or contact us for further details: **permissions@emeraldinsight.com**

Downloaded by TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES At 22:54 01 November 2016 (PT)