Path-based Keyword Search over XML Streams

Savong BOU†, Toshiyuki AMAGASA††, and Hiroyuki KITAGAWA††

† Graduate School of Systems and Information Engineering, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan
†† Faculty of Engineering, Information and Systems, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan
E-mail: †savong.bou@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp

Abstract Recently, a great deal of attention has been focusing on processing keyword search over XML and XML streams. The keyword search is simple and provides a user-friendly way of retrieving required data from an XML data. Though its popularity, there is a concern over its efficiency. For this reason, several methods have been proposed to enable keyword search over XML streams. However, most of them primarily aim at processing pure keyword search where only keywords are allowed as a query. However, in many cases, there is a demand to combine keyword search and path-based query. To address this problem, we propose a method to integrate XPath and keyword search so that we can combine XPath- and keyword-based query, thereby making users possible to express their search demands in more specific ways. The experimental results show that the proposed scheme can process queries over XML streams practically.

Key words XML Streams, Keyword Search, XPath, XQuery

1. Introduction

Extensible Markup Language (XML) [1] is a standard for representing semi-structured data. XML has been used in various applications, because it is powerful and versatile while it is simple.

In many applications, information represented using XML is exchanged in real-time. This kind of information is called XML streams [9]. Several applications of XML streams have emerged recently, such as web services, sensor networks, etc. In such applications, it is often required to filter necessary data out of incoming XML streams, and, typically, such requests are represented in terms of XPath expressions [9].

In the meantime, a large number of web services, such as blogs, news sites, and podcast hosts, are currently disseminating their contents in the form of streaming XML data. The variability and heterogeneity of those sources make the usage of traditional querying schemes, which are based on path-based query language [9], cumbersome for end users, because those languages require precise knowledge about the underlying schemas of XML data being queried in order to formulate meaningful queries. On the other hand, keyword search [4, 6–8, 10, 12–14], where only few keywords are used to query XML data, provides a simple and effective query paradigm. For example, if one wishes to retrieve partial XML data related to XML query processing, he only needs to put three keywords, "XML", "query" and "processing" rather than a complicated XPath-based query. For this reason, keyword search over XML data has been studied well, and many XML database systems support such functionality. However, those engines support keyword search over XML data, which are permanently stored and can be indexed for efficient retrieval, and few research works address keyword search over XML data in streaming environments [6, 8, 13].

In [8, 13], they proposed a scheme to process multiple keyword-based queries over XML streams. It offers a good compromise between performance and memory usage, which makes it the best choice in real world scenarios. Besides, [6] proposed a technique based on refining a keyword query to a query graph being a sub-graph of corresponding DTD document so that their stack-based algorithm can process XML stream and manage the buffer of search results in a single pass efficiently using their new search heuristic.

It is worth to mention that, in many application scenarios, there is a strong need to make keyword search against some specific parts of XML data whose structures constantly change and are unknown or little known. Taking a bibliographic XML data for example, one may make a query only on abstract, but not to elsewhere. However, since keywords in keyword search can appear either as label(XML element) or textual value, and can carry multiple and different meanings, it is hard to express the exact search intention only with keywords. In particular, it is hard to specify which parts of XML data the keyword search should be applied to.

The above problem can be solved by combining XPath-based query with keyword search. The XPath-based query will be used as a mean to specify which part of XML data the keyword search should be applied to, and the keywords are used to specify the user's demand for the query results. It should be noticed that the combination of XPath- and keyword-based queries is beneficial to the users, because they can exploit the benefits from both query styles, that is, one does not need to fully understand the structure of the documents being queried, while having the freedom to limit the parts of the documents to be retrieved in terms of an XPath expression. However, to the best of authors' knowledge, no research has been made on this type of query in a streaming setting so far.

To address this problem, we propose a scheme to process XPath queries combined with keyword search over XML streams. More precisely, we extend NFA model to support XPath-based keyword search. We also extend NFA-based YFilter [5] with the method used in CKStream [8].

2. Preliminaries

First, we would like to explain some preliminaries related to our work.

XPath XML Path Language (XPath) [9] is an expression language that allows us to locate specific XML fragments in a given piece of XML data. More precisely, an XPath expression, `P`, is defined as the following grammar [9]:

```
P ::= /N | //N | PP
N ::= E | A | text(S) | *
```

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`.

XML Keyword Search is a style of XML data search, where a set of keywords are given as the input, and the set of ranked XML fragments that match with the query specification is returned as the query result. In [4, 8, 13], the syntax of keyword search is defined as follows.

[Definition 1] An XML keyword search `Q` is a set of search terms $(t_1, .., t_m)$. Each query term is of the form: `l::k`, `l::`, `::k`, or `k`, where `l` is a node label and `k` a keyword. A node n_i satises a query term of the form:

- `l::k` if n_i's label equals `l` and the tokenized textual content of n_i contains the word `k`.
- `l::` if n_i's label equals label `l`.
- `::k` if the textual content of n_i contains the word `k`.
- `k` if either n_i's label is `k` or the tokenized textual 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, which is 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 this data source, we observe that the word `"book"` appears as both XML element and textual value. Therefore, if the user issues keyword search, q_3, `"book author::Hiroki"`, the keyword `book` matches both XML element `"book"` whose ID is 2 and textual value of XML element `"title"` whose ID is 8 because this keyword is in the form of `k` as mentioned above.

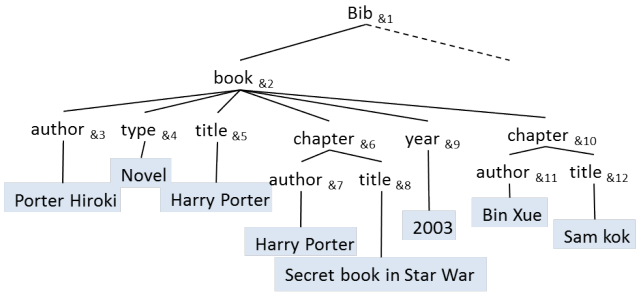


Fig. 1: An example of XML data

Combining XPath with keyword search: it is natural to extend XPath so that it can deal with keyword-based queries. More precisely, keyword search can be used to specify a query predicate in an XPath expression. In fact, XQuery and XPath Full Text 1.0 [2] is a W3C standard for that purpose. Since our objective is to combine keyword search with XPath-based queries, we partially borrow the syntax from it. Here is the basic syntax.

```
/XPath[ftcontains(keyword-search query)]
```

where `XPath` is an XPath expression and `ftcontains` is a dedicated function to specify a keyword search query according to [4,8,13]. Note that the XPath and keyword-search query are connected by a descendant axis.

Node Relatedness Heuristics in XML keyword search, for a set of given keywords, it is essential to decide which XML fragments are most eligible to be query results. For this reason, many heuristics have been proposed [4,10,12,14]. Among these, LCA (Lowest Common Ancestor) is known to be the fundamental method, where the lowest XML fragments that subsume the entire keyword set and the fragments are identified. Subsequently, to improve LCA, many variants have been proposed.

[Definition 2] SLCA (Smallest Lowest Common Ancestor) [14]: 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 $LCA(n_3, n_4)$ descendant of $LCA(n_1, n_2)$.

Node $v \in d_i$, such that $v \in LCA(n_1, n_2)$, $v \in SLCA(n_1, n_2)$.

However, there are some criticisms against SLCA heuristic, for its answer is too compact, and the smallest subtree is not always the correct answer of keyword search. As a result, some accurate answers are discarded. For example, we have a keyword search `"author::Porter title::"`, which is to search for the `title` of any publication written by `author "Porter"`. Based on SLCA heuristic, the subtree rooted at node `chapter` ID 6 is returned. However, the subtree rooted at node `book` ID 2 is not returned as answer eventhough it is also the correct result of this query. To complement such loopholes, MLCA heuristic [11] has been proposed. MLCA heuristic takes the relationship between pairs of all keywords in the query into consideration; consequently, each result under MLCA heuristic is a pattern match, in which every two nodes are meaningfully related. Therefore, with MLCA heuristic, the answers of the above query are sets of all matched nodes in subtrees rooted at node `chapter` ID 6 and `book` ID 2. MLCA is defined as follow:

[Definition 3] MLCA (Meaningful Lowest Common Ancestor) [11]: 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 $A, B \subseteq N$, where A is comprised of nodes of type α, and B is comprised 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 $MLCASET(A, B)$.

3. Existing Approaches

3.1 YFilter

YFilter [5] is an XML filtering system that provides real-time, fast matching of large numbers of queries, containing constraints on both structure and content, against both static XML data and XML streams. The key innovation in YFilter is that it generates a single Nondeterministic Finite Automation (NFA) from all the input-queries. YFilter also provides better structure matching and additional benefits including a relatively small number of machine states, incremental machine construction, and ease of maintenance.

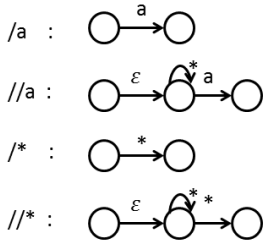


Fig. 2: Basic NFA Location Step

Figure 2 shows the NFA fragments of basic location steps. There is a transition from one state to another state via a directed edge representing a transition. 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.

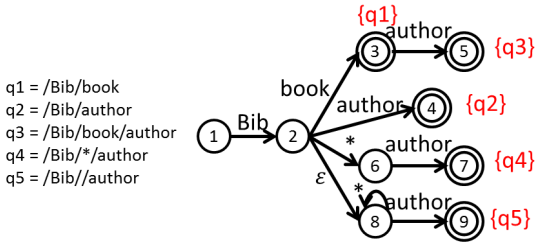


Fig. 3: XPath queries and a corresponding NFA

Figure 3 shows an example of such a Non-deterministic Finite Automation (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.

When processing an XML data, it is parsed with a SAX parser, which is an event-based XML parser; whenever it reads a new XML constructs, such as start- and end-tags, text content, etc., it raises a corresponding event and is notified to the application program. When YFilter receives a start element events, it triggers a state transitions in the FSMs, while an end element event is received, YFilter must backtrack to previous states. A run-time stack is used to track the active and previously processed states.

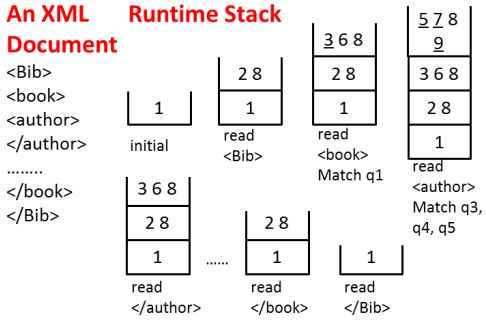


Fig. 4: An example of query processing in YFilter

Figure 4 shows a running example of the run-time stack, which process queries in Figure 3 by XML data in Figure 1. The content of stack is a set of active state ID. When receiving a start element event, it follows 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 [8] is an algorithm to process multiple keyword

search over XML streams. It relies on parsing stacks and query indexes specially designed to allow the simultaneous matching of terms from different queries. CKStream also relies on a SAX parser.

Parsing Stack: when parsing an XML stream, each visited XML node, e.g., element or text, is associated with an entry in a stack, and that entry handles the following information: (1) the label of the element of the entry; (2) a bitmap called CAN_BE_SLCA , which contains one bit for each query being evaluated, (3) a set used queries containing the IDs of the queries whose terms include keywords and (4) keywords. Each entry is popped from the stack when its corresponding node and all its descendants have been visited.

Query Index: To efficiently deal with a large number of queries, the algorithms exploits query indexes for efficient query look up. Each index entry represents a query term and refers to queries in which this term occurs. It also differentiates between structural (label) and non-structural (value) query terms. A sample query index created from queries q_1, q_2, and q_3 above is shown in Figure 5.

Fig. 5: Example of Query Index

Query Bitmap: Stores information about which keywords from each query have occurred in an XML node, which is kept using a bitmap stored on the stack entry corresponding to such a node. This bitmap is called query bitmap and each of its bits is associated with one unique search term from each query. The information stored in `query bitmap` is similar to that of `query index`, but they are used differently. A sample of query bitmap from queries q_1, q_2, and q_3 above is shown in Figure 6.

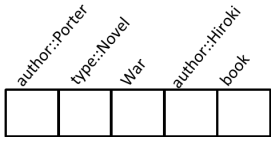


Fig. 6: An Example of Query Bitmap

When a startelement(tag) comes, it searches the query index for the current node's label, obtaining the positions associated with the entry corresponding to this label. It sets to true the bit position associated with this occurrence in the query bitmap of the new stack entry. Similarly, when character() comes, it searches the query index and sets true to the corresponding bits. Notice that terms of the form `l::k` are handled in this function and receive a specic treatment. Finally, when an endelement(tag) comes, it looks for complete queries , i.e., those in which all search terms are satised by the node being processed (or by its decedents).

4. Proposed Scheme

4.1 Proposal Overview

We extend NFA-based YFilter [5] with keyword-based CKStream [8] in an attempt to make the filtering engine supports XPath query, keyword search query, and XPath-based keyword search. More precisely, we extend the NFA-based finite state machine in YFilter by adding the data structures in CKStream, such as a set of used queries and query bitmap, 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 [12]. MLCA returns a set of nodes that match queries.

4.2 Extension of NFA Model in YFilter

In YFilter, a state transition occurs only when an XML element is read, but, to support keyword search, a state transition is also needed for text content. For this reason, we modify the NFA model as follows. The edges with the labels of the form `"text() con"` represent state transitions corresponding to text nodes, that are enabled when text node contains the specified keywords. In addition, each accepting state in the extended NFA contains the position of bit inside the query bitmap. Queries are matched by checking if all bits corresponding to their respective queries are matched. The basic extended-NFA location steps are shown in Figure 7.

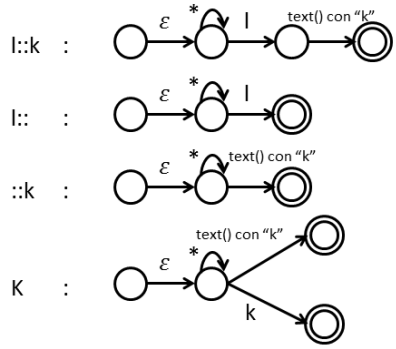


Fig. 7: Basic Extended-NFA Location Step

4.3 Combining YFilter and CKStream

By using the extended NFA model, our proposed work constructs an extended-NFA corresponding to a set of XPath-based keyword search. Similar to YFilter and CKStream, the proposed scheme based on SAX parser. In addition, two data structures, query_bitmap and used_query, are borrowed from CKStream to process keyword queries; they are used whenever there is a push of an entry into the stack. When there is a trigger to pop an entry from stack, queries are evaluated to be matched or not by looking at bits in query_bitmap corresponding to each used queries from set used_query.

The details of our proposed algorithm are shown in Algorithm 1 below. There are 3 main functions: **Callback Function Start of Element**, **Callback Function Text**, **Callback Function End of Element**.

Callback Function Start of Element: When the startelement(tag) is called, this function is invoked. All bits in the query bitmap are set to false by default. Then startelement(tag) is processed against the single extended-NFA. Then all the newly-active states are checked one by one. If any of those states are accepting states, the query IDs and the positions of bits in the query bitmap will be obtained. The query IDs are put in the set of used queries and the bits of the query bitmap are set to true at the obtained bit positions. If no accepting state exists, all bits in the query bitmap remain false and the set of used queries is blank. Finally, the set of active states, query bitmap, and set of used queries are inserted into an entry, which later pushes into the stack.

Callback Function Text: When the character() event is called, textual content is split into tokens. Then each token is processed by the single extended-NFA. Following the same procedure as mentioned in **Callback Function Start of Element**, the information that is obtained from the accepting states are used to update the entry at the top of the stack (the entry of parent nodes in the stack). In **Callback Function Text**, no new query bitmap nor new set of used queries are created; and therefore, no new entry is pushed into the stack.

Callback Function End of Element: When an endelement(tag) is called, the entry is popped out from the stack. Then all query IDs in the set of used queries in the popped entry are checked one by one. For each query ID being processed, all bits in the query bitmap of the corresponding query are checked. If they are all true, the corresponding query matches, and the results are returned. If not all bits of the corresponding query are true, then that corresponding query IDs is added to the set of used queries of the top entry of the stack and all bits in the query bitmap of the popped entry are used to update the query bitmap of the top entry by using `"OR"` operator.

We show how our proposed method works by running the two queries below against the XML data shown in Figure 1. The detail run is shown in Figure 9.

Q1: `//book[ftcontains(author::Porter type::Novel)]`

Q2: `/bib/book/chapter[ftcontains(author::Porter War)]`

These queries contain three unique keywords, so the query_bitmap will be of size 3 where t_1, t_2 and t_3 are the position of each keyword in the query bitmap. By using the extended-NFA model, from the two queries, a single NFA is constructed as shown in Figure 8.

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}
3: `tn := *top(S)` tn points to the top entry in the stack
4: **while** `q` ∈ `sn.used_queries` **do**
5: | let $j_1,\cdots, j_N$ 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**

As shown in Figure 9, when new XML data, startDocument(), comes, the initial state is initialized and pushed into the stack. When receiving an event startelement(tag), the system follows the same rule as that in YFilter to get the next states and pushes into the stack. If the element is associated with textual content, on receiving character() event, the system follows the same procedure to get the newly active states. If none of the newly-active states are accepting states, ablank query_bitmap and set used_query are created. If one or more newly-active states are accepting states, the positions of the bitmap are obtained, and bitmaps of the newly-created query_bitmap are set to true to the corresponding obtained positions. The IDs of queries obtained from the accepting states are added to the set used_query. Then a new entry is pushed into the stack.

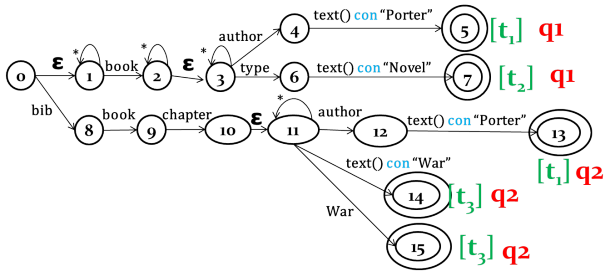


Fig. 8: A Single NFA

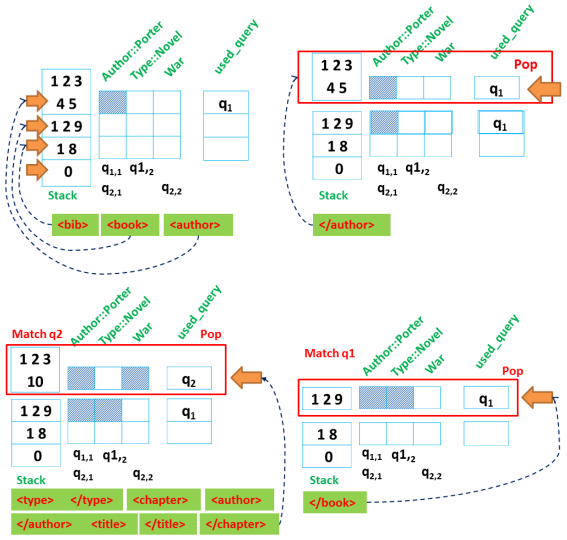


Fig. 9: A Running Example Proposed Method

When receiving an event endelement(tag), the entry is popped of the stack. Then it checks the set used_query of the popped entry. If the used_query is not empty, IDs of each query are used and it checks the bitmaps of the corresponding queries in the query_bitmap of the popped entry. If all bitmaps of any queries are true, the respective queries match. Otherwise, they do not match. If none of queries match, all bitmaps of the query_bitmap are sent and combined with entry of the top stack by using "OR" operator. The set used_query of the popped entry is added up to that of the top entry.

5. Experimental Evaluation

5.1 Setup

The algorithm is implemented using Java on the existing YFilter [5] and the SAX API from Xerces Java Parser in an Intel Core 2.33GHz computer with 2GB of memory in Windows XP Service Pack 2. All data structures, query bitmaps and sets of used queries, are kept entirely in memory.

We use two types of datasets, synthetic data and real data. Our synthetic data is generated by the xmlgen of XMark [3]. Our real datasets are DBLP-biographic information on major computer science journals and proceedings, Mondial-world geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database. Details of the datasets are presented in Table 1.

Table 1: All datasets used in the experiments

Dataset	Element	Attributes	Max-depth	Avg-depth
XMark	333 millions	333 millions	5	3
DBLP	3,332,130	404,276	6	2.90
Mondial	22,423	47,423	5	3.59

We generate XPath-based keyword search from the above datasets. Since there is a performance impact when searching for different kinds of keywords in our system, we separately generate sets of queries which contains only keywords in the forms of `l::`, `l`, `l::k`, `::k` and mixed of the four forms. For example, we could generate: //regions[ftcontains(::begin ::caius)], whose keywords are in the form of `::k` from XMark dataset. Moreover, since the same keywords can appear in different queries, we devide 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.

We measure the elapsed time, average memory usage, and number of extended-NFA states for processing all XML data on a given stream. This includes the time spent to create the NFA, query bitmaps, and the set of used queries.

5.2 Varying the Number of Queries

We vary the number of queries from 1, 10, 100, 200, 400, 600, 800, and 1000 and investigate the impact on the performance of our algorithm. 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 10, 11, 12, 13, 14, and 15. This is because the system needs to process each single query and output the result of each queries. Figure 10 shows that the number of NFA states increases when the number of queries increases from 1 to 100, but when the number of queries increases from 100 to 1000, the number of NFA states becomes constant at 31 because same keywords

appear at different queries very frequently and the number of unique labels is only 31 in DBLP. Such cases also happen to Mondial and XMark dataset as shown in Figures 11, and 12. We could not generate more than 100 queries in which the same keyword appears only once in a particular query in the same set of queries. As a result, the increasse of memory usage and the decrease of throughputs do not change much when the number of queries increases.

5. 3 Varying the Number of Query Terms

We investigate the impact on the performance of our algorithm when we increase the number of keywords from 2, 4, and 6. 6 keywords are a reasonable limit when one uses to specify the query [8]. We randomly generate sets of queries whose same keywords can appear in more than one query. With these sets of queries, we observed that though the number of keywords increase, the memory usage, number of extended-NFA states and throughputs do not change much between 2, 4, and 6 keywords. These cause by the more frequency that same keywords appear at different queries as shown in Figures 10, 11, and 12.

Next we generate sets of queries in which the same keywords appear at only one query in the same sets. Then we investigate the impact that the increase in unique keywords in the queries on the performance of the algorithm. As expected, as the number of unique keywords increase, the number of NFA states also increase. As a result, the memory usage increases and the thoughputs decrease significantly as shown in Figures 13, 14, and 15. Though memory usage and throughputs of the algorithm have some degradation when the number of queries and the number of keyword words increase, the algorithm scales well with such increase.

6. Related Works

XML Keyword Search: Using keywords to query XML databases has been extensively studied. XSEarch [4] adopts an intuitive concept of meaningfully related sets of nodes based on the relationship of the nodes with its ancestors. XRANK [10] presents an adaptation of Google's PageRank to XML data for computing a ranking score for the answer trees. The work in [8] and [13] took the first steps towards processing keyword-based queries over XML streams.

XML Streams: Recent research on XML streams (dissemination, ltering and routing) aims at building large-scale, distributed systems [4, 5, 8, 9]. Most of those are concerned with performance aspects and optimize the ltering process [9], the message delivering and the routing network. YFilter [5] uses NFA-based implementation to improve structure matching and number of machine states are small. In order to deal with multiple XML schemas, two approaches are very common: to use query rewriting [15] and global schemas [11].

7. Conclusion

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

We evaluate them by some experiments on both synthetic and real datasets. 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, XPath-based keyword search, to multiple XML streams, which useful information can be obtained when informations from different sources are combined at real-time.

Acknowledgement

This work is partly supported by the collaborative research (Fujitsu Lab. Ltd., CPE25149), the Okawa Foundation research grant, and JSPS KAKENHI (25330124).

References

[1] Extensible markup language. *www.w3.org/XML/*, 2013.

[2] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient xml search with complex full-text predicates. In *SIGMOD*, pages 575–586, ACM New York, NY, USA, 2006.

[3] R. Busse, M. Carey, D. Florescu, M. Kersten, I. Manolescu, A. Schmidt, and F. Wass. Xmark-an xml benchmark project. *http://www.xml-benchmark.org/*, 2013.

[4] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: a semantic search engine for xml. In *In Proc. Of VLDB*, volume 29, pages 45–56, 2003.

[5] Y. Diao and M. J. Franklin. High-performance xml filtering: An overview of yfilter. In *IEEE*, pages 41–48, 2003.

[6] M. Gawinecki, F. Mandreoli, and G. Cabri. Keyword search over xml streams: Addressing time-stamping and understanding results. pages 371–382, U. of Modena, 2008.

[7] V. Hristidis and N. Koudas. Keyword proximity search in xml trees. *IEEE Trans.*, 18:525–539, 2006.

[8] C. Hummel, S. da Silva, M. Moro, and Laender H.F. Multiple keyword-based queries over xml streams. In *ACM*, pages 1577–1582, ACM New York, NY, USA, 2011.

[9] A. Gupta J. Green and M. Onozuka. Processing xml stream with deterministic automata and stream indexes. *ACM*, 29:752–788, 2004.

[10] F. Shao L. Guo, C. Botev, and J. S. Xrank: ranked keyword search over xml documents. In *In Proc. Of SIGMOD Conf.*, pages 16–27, ACM New York, NY, USA, 2003.

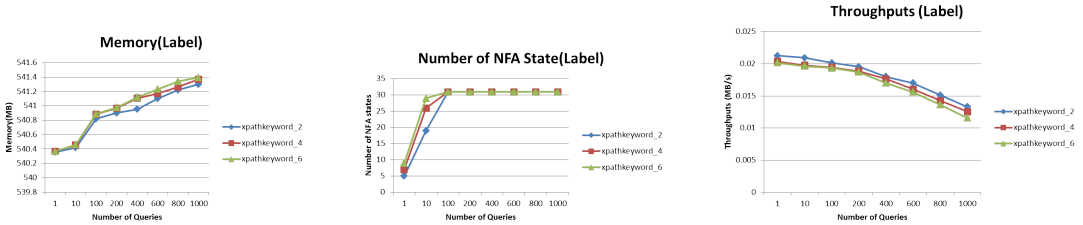
[11] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, volume 30, pages 72–83, 2004.

[12] Z. Vagena, L. S. Colby, F. Ozcan, A. Balmin, and Q. Li. *On the Effectiveness of Flexible Querying Heuristics for XML Data*, volume 4704. 2007.

[13] Z. Vagena and M. Moro. Semantic search over xml document streams. In *DATAX*, 2008.

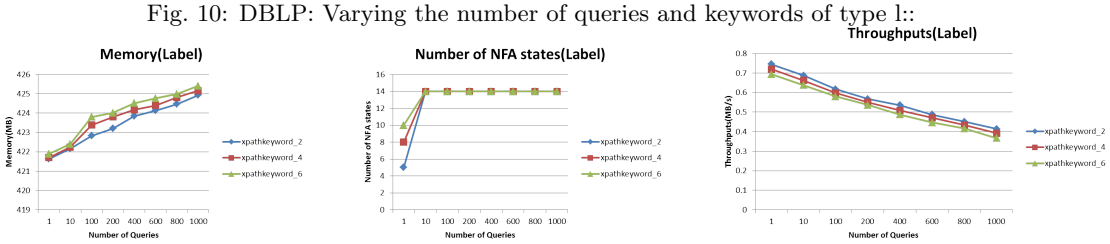
[14] Y. Xu and Y. Papakonstantinou. Efcient keyword search for smallest lcas in xml databases. In *In Proc. of SIGMOD Conf.*, pages 527–538, ACM New York, NY, USA, 2005.

[15] C. Yu and L. Popa. Constraint-based xml query rewriting for data integration. In *In Proc. of SIGMOD Conf.*, pages 371–382, ACM New York, NY, USA, 2004.



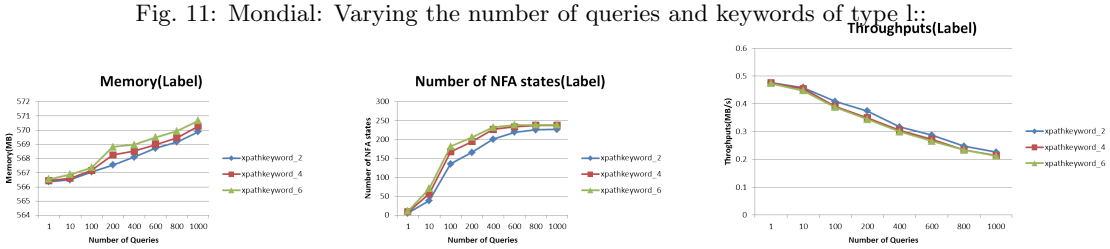
(a) Memory usage (b) Number of NFA states (c) Throughputs

Fig. 10: DBLP: Varying the number of queries and keywords of type l::



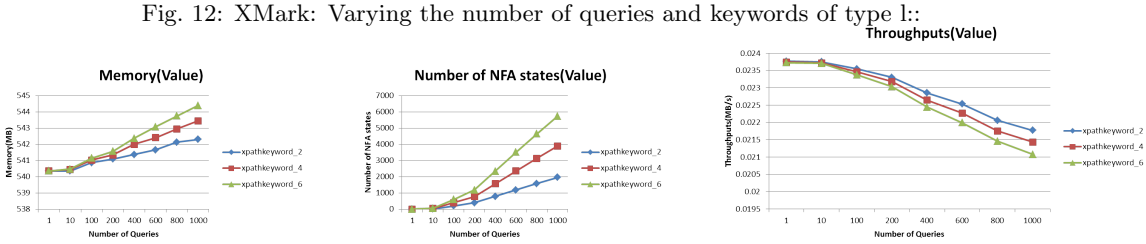
(a) Memory usage (b) Number of NFA states (c) Throughputs

Fig. 11: Mondial: Varying the number of queries and keywords of type l::



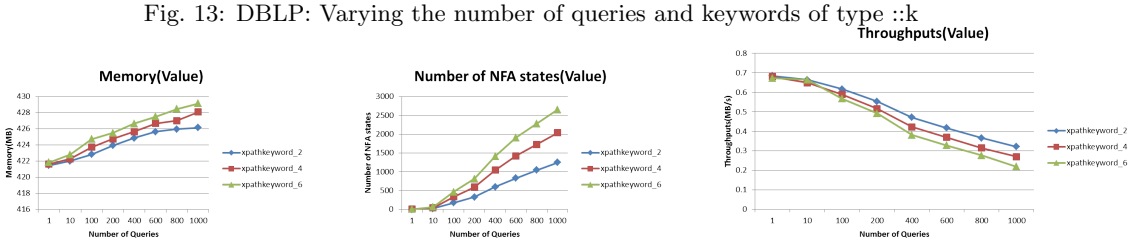
(a) Memory usage (b) Number of NFA states (c) Throughputs

Fig. 12: XMark: Varying the number of queries and keywords of type l::



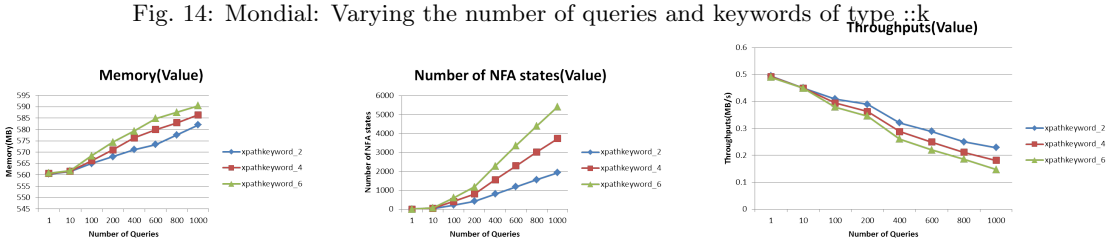
(a) Memory usage (b) Number of NFA states (c) Throughputs

Fig. 13: DBLP: Varying the number of queries and keywords of type ::k



(a) Memory usage (b) Number of NFA states (c) Throughputs

Fig. 14: Mondial: Varying the number of queries and keywords of type ::k



(a) Memory usage (b) Number of NFA states (c) Throughputs

Fig. 15: XMark: Varying the number of queries and keywords of type ::k