A Study on Keyword Search over Structured and Semi-structured Data Streams

March 2017

Savong Bou

A Study on Keyword Search over Structured and Semi-structured Data Streams

Graduate School of Systems and Information Engineering

University of Tsukuba

March 2017

Savong Bou

Abstract

With the rapid growth of real-time information from various sources, continuous query processing over data streams has become increasingly important. To extract information from streams, keyword search is considered to be useful because it allows users to issue queries without having detailed knowledge about the streams, e.g., schema, data types, etc., as well as query languages, such as SQL. Besides, streams are of different types with respect to the used data format, and relational streams and XML streams are the most popular ones. To enable keyword search, different techniques need to be developed depending on the characteristics of streams being processed. In this dissertation, we propose scalable and efficient ways to enable keyword search over XML and relational streams by addressing the following major problems: 1) quality of search results in keyword search over XML streams and 2) scalability issue when processing long queries over relational streams. For the first problem, we observe that there are many cases where one would like to make keyword search on partial XML data, e.g., keyword "XML" should appear in the abstract, while existing approaches do not support such XPath-enabled keyword search over XML streams. To address this problem, we propose a method to integrate XPath-based search and keyword search over XML streams by integrating existing YFilter with CKStream. As a result, we enable efficient filtering over XML streams according to user-specified filtering conditions consisting of XPath expression and query keywords. For the second problem, we observe that the existing approaches that exploit Candidate Networks (CNs) do not scale enough in particular when the number of query keywords and/or the maximum size of query results is large due to the exponential blowup in terms of the number of CNs. To cope with this problem, we propose a novel query processing technique exploiting a new data structure called *MX-structure* (maximal-sharing structure), where CNs are consolidated as much as possible to generate efficient query evaluation. The experimental results prove that the proposed methods perform much better than the existing approaches.

Acknowledgements

A major research project is never the work of anyone alone. The contributions of many different people, in their different ways, have made this possible. Similarly, I would never have been able to finish my dissertation without the guidance of my committee members, help from friends, support from my family.

First and foremost I would like to express my sincerest gratitude to my supervisor, Professor Hiroyuki Kitagawa, who has supported me throughout my thesis with his patience, motivation, enthusiasm, and immense knowledge. I attribute the level of my Doctoral degree to his encouragement and effort and without him thesis, too, would not have been completed or written. I could not have imagined having a better advisor and mentor for my Doctoral Study.

I also owe my deepest gratitude to my co-supervisor, Professor Toshiyuki Amagasa, for making this research possible. His excellent guidance, caring, patience throughout the research project provide me with an excellent atmosphere for doing research. Also, he has spent his precious time listening to my every single presentation and research-related problem, as well as his pain-staking effort in proof reading the drafts, are greatly appreciated. Indeed, without his precious guidance, I would not be able to put the topic together.

I would also like to thank Assistant Professor Yasuhiro Hayase, Assistant Professor Chiemi Watanabe, and Assistant Professor Hiroaki Shiokawa. They are so kind and very supportive, and they helped me a lot from various aspects.

Besides, I would like to thank the following committee members of this doctoral dissertation: Professor Hiroyuki Kitagawa, Professor Kazuhiko Kato, Professor Koichi Wada, Professor Tetsuji Satoh, and Associate Professor Toshiyuki Amagasa, for their insightful comments and encouragement. Their questions and comments greatly encouraged me to widen my research from various perspectives, and most importantly improve the quality of this dissertation.

I would also like to offer my special thanks to Ms. Yumiko Hisamatsu and Ms. Tetsuko Sato, the precious secretaries in KDE. They have helped and supported me a lot in various academic-administration-related works. I thank my fellow labmates, seniors, and juniors in KDE, who have made friend with me, helped me in various ways of my time here.

I would also like to thank my parents and brothers for their unconditional supports.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the current trends of Cyber Physical Systems [84, 85], Internet of Things [86, 87], explosive usage of social medias [129–132], etc., the volume of real-time information from different sources has been explosively increasing. Such information is called data streams. Different from permanently-stored data, data streams are very diverse in contents, fast changing, continuously and rapidly arriving, and very unpredictable. Moreover, their volume is very huge, possibly infinite, which means it is impossible to store all information from data streams into disks. Therefore, to make use of data streams, it requires fast and efficient processing.

Data streams are very useful and have been applied to a wide variety of fields, such as sensor networks and network traffic analysis, business and financial trackers, Web logs and Web page click streams, cyber attack detection, telecommunication calling records, engineering and industrial process, etc. So far, there are a lot of data streams processing frameworks, such as STREAM [133, 134] (STandford stREam datA Manager), SAP event stream processor [135, 136], Photon [137], and NiCT [141]/NICTER [138–140].

Therefore, processing of data streams is very vital and has been a very hot research topic. There are a lot of processing techniques regarding data streams, such as sampling of data streams [142], incremental computation of streams [145], estimating moments of streams [143], complex event processing [144], window-aggregate [146], window-join [147], query processing of data streams [25–27, 35, 37, 41, 44, 49, 51, 52, 59–61, 89–105, 107–120, 122–124, 127, 128], etc. Query processing is considered to be one of the most useful techniques in retrieving important information from data streams.

In query processing of data streams, different techniques should be exploited for query formulation and its processing depending on the targeted data streams. Data streams are of various forms ranging from pure texts to structured data streams according to their usage. Typically, they can be grouped into three categories: 1) structured data streams that are highly organized and conformed to strict schema (e.g., relational streams [59–61]), 2) semi-structured data streams that do not conform with relational databases or other forms of data tables but contain tags or markers to enforce hierarchies of records (e.g., XML streams [25, 35, 37, 41, 44, 49, 51, 52], JSON streams [106], RDF streams [107–111]), and 3) unstructured data streams that are the opposite of structured data streams and do not have pre-defined data model or format (e.g., text streams [122–124, 127, 128]). XML and relational streams are two of the most popular data streams that have been extensively used for the last decade.

Extensible Markup Language (XML) [2] is a popular language for exchange of data over

the Web and has been used in various applications because it is powerful and versatile while simple. In many applications, information represented using XML is exchanged in real-time. This kind of information is called XML streams. Several applications of XML streams have emerged recently, such as Web services [148], sensor networks, message routing [149], 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 [25].

Similarly, relational data [58] has been a leading administrative-data-centric application for decades and has been extensively used in countless applications because of its powerful data storage and retrieval technology. Due to the explosive increase in the number of real-time information sources, it has become very common to investigate interesting information that is interrelated from various information sources interconnected in the form of relational model. Such interconnected stream data sources can typically be modeled as relational streams, where structured records (relational tuples) are transmitted. Recently, a lot of relational streams processing frameworks have been developed, such as NICTER [138–140], SAP event stream processor [135, 136], and STREAM [133, 134]. Therefore, extracting needed information from relational streams has become very crucial, which can be done by using standard query language like CQL [89, 90].

This dissertation focuses on query processing over XML and relational streams. To retrieve information from XML and relational streams, standard query languages, like CQL and XPath, are commonly used. However, these query languages are not suitable for naive users because users have to know the specification of query languages and the detail about the schema of streams. Compared to such conventional query languages, keyword search [26, 27, 29, 33, 54, 58, 62, 71, 75, 75] is considered to be a better solution due to its simplicity and its user-friendliness. Therefore, this dissertation focuses on keyword search over XML and relational streams.

Keyword search over XML streams [26,27,99–105] is a search technique where the input queries are just a set of keywords. These keywords are used to evaluate over XML streams, where XML elements and their textual values continuously arrive. The answers of keyword search over XML streams are XML sub-trees that contain all keywords of any query.

For keyword search over relational streams [59–61], all keywords of the input queries are continuously evaluated against relational streams. The answer of this search framework is a tuple or a set of connected tuples from various tables, which contain all keywords of any query. The number of connected tuples in the search result can be huge. Therefore, a predefined parameter, T_{max}, is used to limit the maximum number of tuples that are allowed in each search result.

There have been extensive studies of standard query languages, such as CQL and XPath, over XML and relational streams, but there have not been many works done to enable the processing of keyword search yet. The survey in Section 3 shows that existing algorithms for keyword search over XML and relational streams greatly suffer from two main challenges: 1) many unwanted results are returned as results to the users for keyword search over XML streams, and 2) filtering performance is very poor when processing longer queries (queries with many keywords) for keyword search over relational streams. The first challenge happens because the search intentions that are expressed in terms of pure keyword search are hard to be interpreted, therefore not well understood by the algorithms. For the second challenge, when processed queries contain many keywords, the number of partial results that need to be kept and instantly evaluated against the future incoming streams are exponen-

tially increased, which cannot be handled well by the existing algorithms.

This dissertation presents two efficient approaches to address the above two problems as follow:

Keyword search over XML streams 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 want to retrieve the abstract containing some keywords. However, since query keywords in keyword search can appear either as labels (XML elements) or textual values 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 to which the keyword search should be applied. This problem becomes worse when the XML data is rich in textual contents.

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 XPath expression.

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 [35] model to support XPath-based keyword search. We also extend NFA-based YFilter [35] with the method used in CKStream [26].

Keyword search over relational streams It is very important that search engines must be able to efficiently process longer queries because a lot of queries that are registered to the real search engines are getting longer. This is because many users have bad experience in using short queries that lead to too many unwanted results. Therefore, using longer queries in an attempt to get more related results has become a popular trend for many users. As reported in [88] by Hitwise in 2009, the average query length to the search engines has been increasing. For example, the ratio of queries containing more than five words has increased by 10% over the years, while that of single keyword queries has decreased by 3%. In addition, needed information may sometimes span across many tables. To retrieve such information, it is required that the results must contain more tuples. For this reason, the parameter which defines the maximum number of tuples allowed in the query result (T_{max}) must be set bigger because if T_{max} is set to too small, it is impossible to get all needed information. Therefore, efficiently processing of data streams with respect to these two parameters (long query and big T_{max}) are very crucial for any capable search engine.

With respect to the above two parameters, the performance of the existing approaches [59, 61] considerably degrades when the number of query keywords and/or T_{max} are increased. The increase of these two parameters causes rapid increase in the number of CNs, which results in a lot of common partial networks remain unintegrated. To exemplify the problem, let

us take TPC-H dataset [57] as an example. When the number of keywords and T_{max} are increased from four to five, the number of CNs increases from 3,600 to 85,803 [61]. Likewise, the total number of edges in the query execution plans exponentially increases from 4,276 to 73,596 in S-KWS [59] and from 7,486 to 222,040 in SS-KWS [61]. (More detailed discussion can be found in Section 5.2.2.) Thus the performance of S-KWS and SS-KWS would deteriorate in particular when dealing with a lot of query keywords and/or large relational streams consisting of many tables.

How can we cope with such exponential blowup of CNs and the complication of query plans? If we consider the edges in CNs, each of them can be associated to be one of the primary/foreign-key relationships between two tables. The number of such relationships is in general small. In other words, we can consider that the edges in CNs are intensively duplicated from the primary/foreign-key relationships in the schema. With the same example above when the number of keywords and T_{max} are increased from four to five, the total number of unique edges in all CNs grows linearly from 1,088 to 3,536. Under this observation, to cope with the problem of CN's exponential blowup, it is possible to consolidate the edges sharing the same primary/foreign-key relationship into one edge when generating a query plan, which leads to great performance improvement. Our algorithm takes into account the above idea. Specifically, a new query plan, called Maximal Sharing structure (*MX-structure*), is proposed to consolidate common edges in different CNs as much as possible.

We evaluate both proposed schemes by extensive experiments on both synthetic and real datasets. The results show that the proposed schemes work well with acceptable throughputs, less memory usage, and good efficiency and utility.

Organization. The rest of this dissertation is organized as follows. Chapter 2 describes all preliminaries that are important to understand this research dissertation. A brief survey about query processing over data streams and all related works to the two proposals are presented in Chapter 3. We present our first proposal, XPath-based keyword search over XML streams, in Chapter 4. The second proposal, keyword search over relational streams, is presented in Chapter 5. Finally, we conclude this dissertation and present future research works in Chapter 6.

Chapter 2

Preliminaries

We would like to brief all preliminaries that are important to understand the whole dissertation. All preliminaries related to XML and relational streams are presented in Section 2.1 and Section 2.2 respectively.

2.1 Extensible Markup Language (XML)

XML [2] is a popular markup language for marking documents by a set of rules for the sake of both human- and machine-readability. The development of XML started in 1996 at the SGML conference in Boston by a group of engineers of the World Wide Web Consortium (W3C). So far, there are several versions of W3C's XML specifications ranging from XML 1.0 first edition to fifth edition. All of them are free open standards.

XML is the successor of the Standard Generalized Markup Language (SGML) [3] that is notorious for its complicated features in learning and parsing. XML has a smaller and simpler syntax while retaining most of the powerful features of SGML. Main features characterized XML are simplicity, extensibility, interoperability, and openness across the Internet. Even though the development of XML has targeted at Web developers, XML certainly has applications beyond it.

XML has becoming popular and has been used in various applications. So far, there are a lot of documents in XML format, such as RSS [4], Atom [5], SOAP [6], and XHTML [7]. Moreover, many administrative applications, such as Microsoft Office [8] (Office Open XML), OpenOffice.org [8], OpenDocument [9], and Apple's iWork [10], use XML as a default format. Other applications of XML are in communication protocols (e.g., XMPP [11]), configuration file of Microsoft .NET Framework [12], etc.

2.1.1 XML Data Model

XML represents semi-structured data by using storage units called entities, which contain either tags or contents. Tag consists of markup and textual data from character data in the document. Tag can contain other tags to describe document's storage layout and logical structure. It also describes the meanings of the contents. There must be end-tag for every start-tag in XML data. The three forms of tag are:

- start-tag, such as <tag>;
- end-tag, such as </tag>;
- empty-element tag, such as <line-break />.

Moreover, XML data must always have only one root element which contains all other elements. Let us look at below XML data of the bibliographic data. In this document, <author> Porter Hiroki </author> indicates that "*Porter Hiroki*" is an author. Similarly, <author>, <type>, and <title> are inside <book> tag, which indicates "*author*", "*type*", and "*title*" of the "*book*".

```
1 <?xml version=''1.0'' encoding=''ISO-8859-1''?>
2   <Bib>
3     <book>
4         <author> Porter Hiroki </author>
5         <type> Novel </type>
6         <title> Cold War </title>
7         <year> 2003 </year>
8         <chapter>
9           <author> Harry Porter </author>
10          <title> Secret book in Star War </title>
11       </chapter>
12       <chapter>
13          <author> Bin Xue </author>
14          <title> Sam kok </title>
15        </chapter>
16    </book>
17    <book>
18         <author> Porter Jame </author>
19         <author> Brad Pitt </author>
20         <year> 2000 </year>
21         <title> Sex and City </title>
22    </book>
23  </Bib>
```

The three main relationships between every elements in XML data are ancestor-descendant, parent-child, and sibling-sibling relationships, which makes tree structure can be easily used to represent XML data. The tree structure of the above XML data is shown in Figure 2.1. This data source is used throughout this thesis.

2.1.2 Parser of XML Data

XML data is in plain text format, so is required special program to access and manipulate it. XML parser is for that purpose. Without the parser, computer cannot process XML data

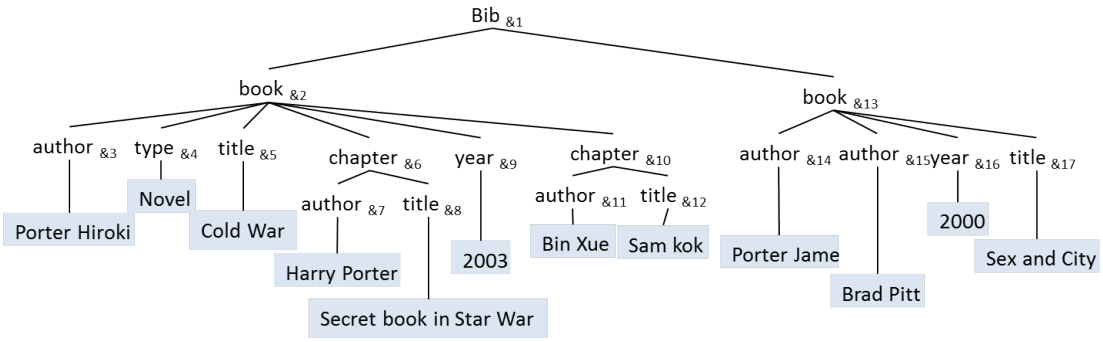


Figure 2.1: An example of XML data

because computer requires instructions on how operations can be done. XML parser provides important information to the computer program on how files can be read.

There are various XML parsers, and they can be categorized into two groups. The first group is referred to the parsers that load the entire XML data into the memory first, then make the in-memory XML data accessible. This kind of parsers, which is called DOM-based parser [13], does not work well with XML streams because XML streams are in theory unbounded. Another group is referred to the parsers that do not load the entire XML data at one time, but load sequentially and continuously send the loaded pieces of XML data in the form of events to the computer program. This kind of parsers is called event-based parser. SAX (Simple API for XML) [14] is one famous event-based parser.

2.1.3 XML Path Language (XPath)

XML Path Language (XPath) [25] is a major element in W3C's XSLT standard, which 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 based on the following grammar [25]:

`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`.

As mentioned above, it is important to know the structure of XML data to issue XPath query. For example if a user wants to get names of *authors* who wrote *books*, published in *year* "*2000*", he may use XPath to get such information. To do this, the user must at least know how to write XPath query and the relationship between all elements *book*, *year*, and *author*. In order to know this relationship, he must know the structure of XML data. Based on the XML data in Figure 2.1, he may issue the XPath ***//book[year=2000]/author*** where "*book*" and "*author*" are XML labels and "*year=200*" is a predicate. Figure 2.2 shows the matched XML element to this XPath. Red circle is the subtree rooted at element *author* ID 14 that matches this query.

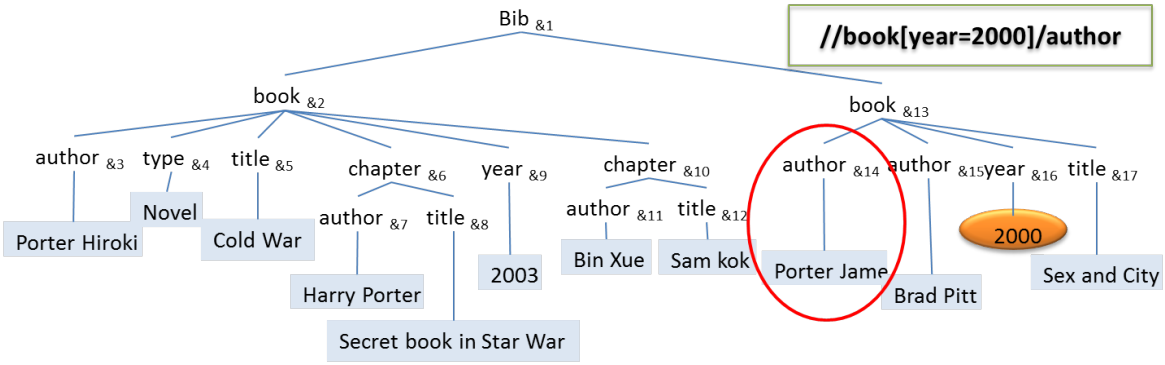


Figure 2.2: Querying XPath in XML data shown in Figure 2.1

2.1.4 XML Keyword Search

XML keyword search [26, 27, 29, 33] is one kind of XML data search, whose input is a set of keywords and a set of ranked XML fragments matching with the query specification is returned as the query result. While processing the queries, some search engines choose to consider only the textual content, while some other search engines consider both the textual contents and the label of each XML node in the searched collection. To better control the document fragments, it is necessary to enable the latter so that the user can explicitly specify constraints on the labels and/or the textual contents. In our work, we adopt this constraints as specified in [26, 27, 29, 33], and thus the syntax of keyword search is defined as follows.

Definition 1. *An XML keyword search [26, 27, 29, 33] 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 satisfies 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.*

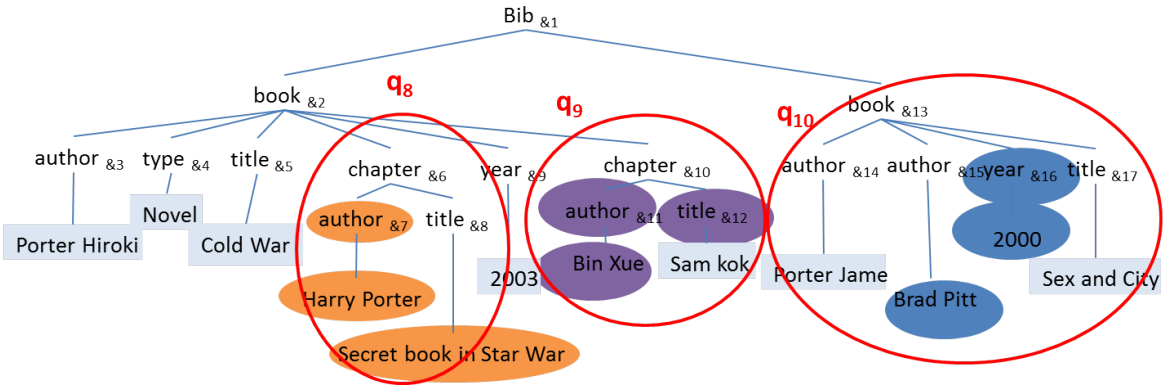
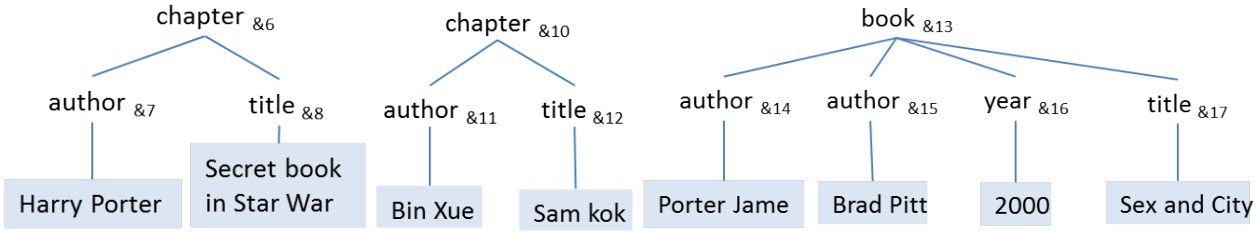


Figure 2.3: Querying keyword search q_8, q_9, q_{10} in XML data shown in Figure 2.2

Table 2.1: Examples of keyword-based queries

Query ID	Keyword search
q_1	author::Porter type:Novel
q_2	author::Porter War
q_3	book author::Hiroki
q_4	author::Hiroki title
q_5	chapter Secret
q_6	type::Novel ::Harry
q_7	::Jame year::2000
q_8	author::Harry ::War
q_9	author::Xue title::
q_{10}	author::Pitt year::2000
q_{11}	book year::2003



(a) The answer of q_8 (b) The answer of q_9 (c) The answer of q_{10}

Figure 2.4: The Answers of keyword search q_8, q_9, q_{10}

To better illustrate the concept of keyword search, we present an example with a fragment of bibliography data source, which is shown in Figure 2.1. 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.

Table 2.1 shows a list of keyword search queries that contain 2 keywords and can be used to search in the XML data in Figure 2.1. To better understand the process of keyword search in XML data, we show the matched-XML elements in the searched XML data, shown in Figure 2.1, for searching queries q_8, q_9, and q_{10} in Table 2.1.

Figure 2.3 shows the matched XML elements to keywords of query q_8, q_9, and q_{10}. The orange ovals are the matched XML elements to q_8, the purple ovals are the matched XML elements to q_9, and the green ovals are the matched XML elements to q_{10}. The red circles are the subtrees in XML data that contain all matched XML elements of each query. Figure 2.4(a), 2.4(b), and 2.4(c) are the answers of q_8, q_9, and q_{10} respectively.

2.1.5 Node Relatedness Heuristics

In XML keyword search, for a set of user's 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 [28, 29, 31–34]. Among these heuristics, 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. *Descendant [36]: We have nodes n_d and n_a belong to the same XML data d_i. n_d is said to be a descendant relationship with n_a if it is a descendant of n_a, denoted as* `descendant(`n_d`,` n_a`)`

Definition 3. *LCA (Lowest Common Ancestor) [36, 151]: For a given query $Q = \{k_1, k_2, \ldots, k_m\}$, an XML document D, and inverted list L_i that stores all nodes directly containing keywords k_i. Let $LCA(v_1, v_2, \ldots, v_m)$ be the lowest common ancestor (LCA) of nodes $v_1, v_2, \ldots, v_m$ where $v_i \in L_i$ $(1 \leq i \leq m)$, then the LCAs of Q on D are defined as $LCA(Q) = \{v$ s.t. $v = LCA(v_1, v_2, \ldots, v_m)$ where $v_i \in L_i$ $(1 \leq i \leq m)\}$*

Definition 4. *SLCA (Smallest Lowest Common Ancestor) [34, 151]: For a given query $Q = \{k_1, k_2, \ldots, k_m\}$, an XML document D, and inverted list L_i that stores all nodes directly containing keywords k_i. Let $LCA(v_1, v_2, \ldots, v_m)$ be the lowest common ancestor (LCA) of nodes $v_1, v_2, \ldots, v_m$ where $v_i \in L_i$ $(1 \leq i \leq m)$, then the SLCAs of Q on D are defined as $SLCA(Q) = \{v$ s.t. $v = LCA(v_1, v_2, \ldots, v_m)$ where $v_i \in L_i$ $(1 \leq i \leq m)$ and $\nexists u = LCA(u_1, u_2, \ldots, u_m)$ where $u_i \in L_i$ $(1 \leq i \leq m)$ s.t.* `descendant(u, v)`$\}$

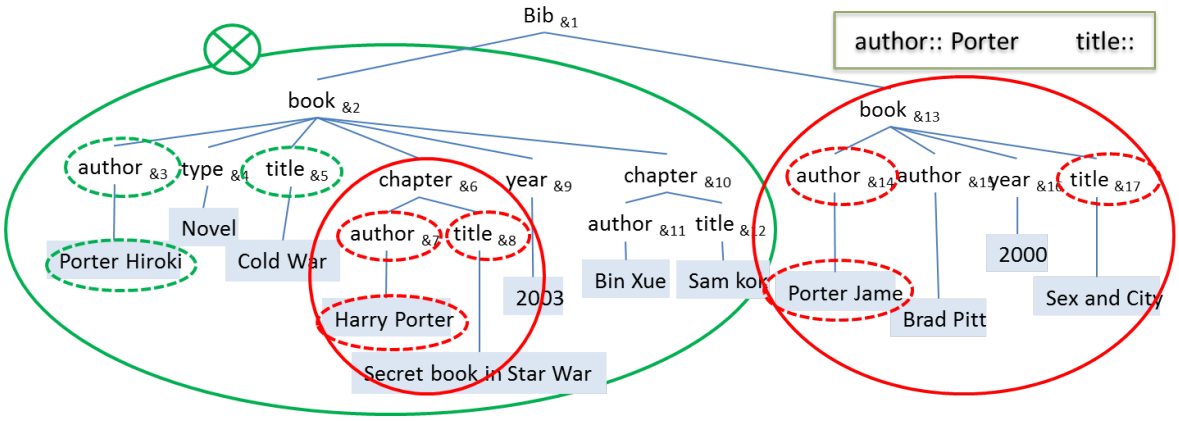
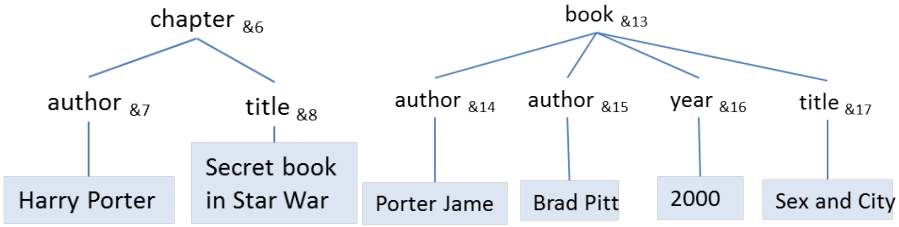


Figure 2.5: Querying keyword search in XML data shown in Figure 2.2 based on SLCA heuristic

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 subtrees rooted at node ***chapter*** ID 6 and at node ***book*** ID 13, which are marked with red circle in Figure 2.5, are returned. However, the subtree rooted at node ***book*** ID 2, which is marked with green circle in Figure 2.5, is not returned as an answer even though it is also the correct result of this query. The reason is that node ***book*** with ID 2 is the ancestor of node ***chapter*** with ID 6, which is a smaller subtree result. As a result, only subtrees, shown in Figure 2.6(a) and 2.6(b), are returned as results.



(a) First answer (b) Second answer

Figure 2.6: The answers of keyword search based on SLCA heuristic

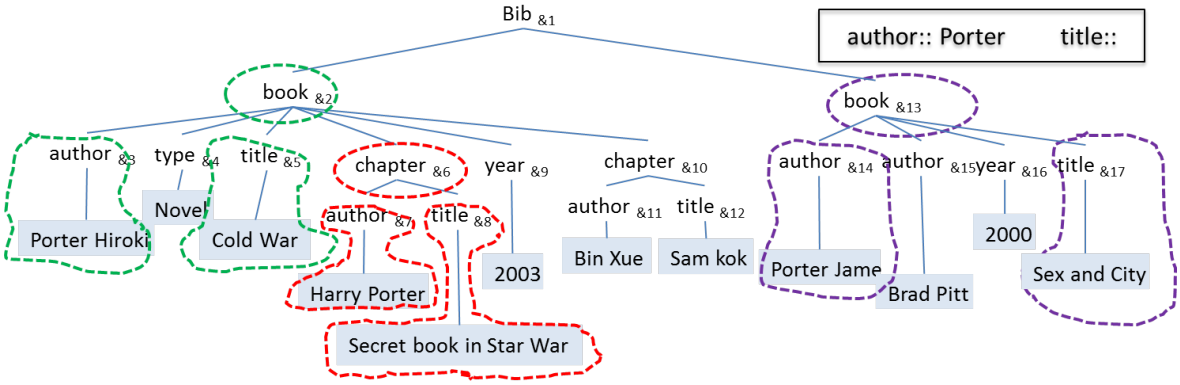
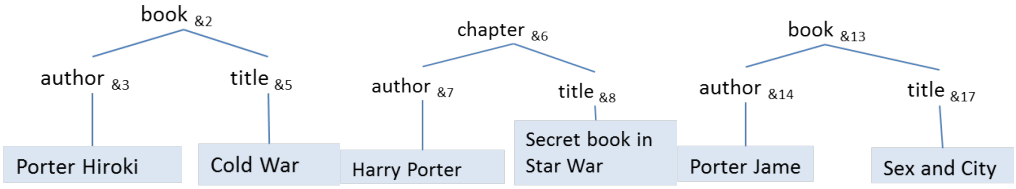


Figure 2.7: Querying keyword search in XML data shown in Figure 2.2 based on MLCA heuristic



(a) First answer (b) Second answer (c) Third answer

Figure 2.8: The answers of keyword search based on MLCA heuristic

To complement such loopholes, MLCA (Meaningful Lowest Common Ancestor) heuristic [36] 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 marked in red circle in Figure 2.7, ***book*** ID 2 marked in green circle in Figure 2.7, and ***book*** with ID 13 marked in purple circle in Figure 2.7. The ansers of this query are shown in Figure 2.8. MLCA heuristic is defined as follow:

Definition 5. *Entity Type [36]: An entity type (or simply type) of a node n in an XML tree is defined as a tag name (label) of n. Two nodes n_1 and n_2 are of the same entity type T if and only if they have the same tag name.*

Definition 6. *MLCA (Meaningful Lowest Common Ancestor) [36]: For a given query Q = $\{k_1, k_2, \ldots, k_m\}$, an XML document D, and inverted list L_i that stores all nodes directly*

containing keywords k_i. Let LCA(v_1, v_2, ..., v_m) be the lowest common ancestor (LCA) of nodes v_1, v_2, ..., v_m where $v_i \in L_i$ ($1 \leq i \leq m$), then the MLCAs of Q on D are defined as MLCA(Q) = {v s.t. v = LCA(v_1, v_2, ..., v_m) where $v_i \in L_i$ ($1 \leq i \leq m$) and $\nexists u$ = LCA(u_1, u_2, ..., u_m) where $u_i \in L_i$ ($1 \leq i \leq m$) and $\exists u_i \neq v_i$ but u_i is of the same type as v_i s.t. `descendant(u, v)`}

We now describe what it means when we say two nodes are meaningfully related to each other in MLCA heuristic. Suppose, we have a keyword search "author, type". Figure 2.9 shows 2 XML fragments, representing structural relationships among keyword nodes of this query in MLCA heuristic. Nodes that contain query keywords are nodes whose labels are "author" and "type". Let nodes *book*, *author*, *type*, and *chapter* represent entities of types *book*, *author*, *type*, and *chapter* respectively. In Figure 2.9(a), node *book* with ID 2 is the LCA of nodes *author* with ID 3 and *type* with ID 4. Therefore, nodes *author* with ID 3 and *type* with ID 4 are meaningfully related to each other by belonging to the same node *book* with ID 2, which is regarded as the Meaningful Lowest Common Ancestor (MLCA) of nodes *author* with ID 3 and *type* with ID 4. However, there is an exception to this second case. As shown in Figure 2.9(b), node *author* with ID 7 is of the same type as node *author* with ID 3, and the LCA of nodes *author* with ID 7 and *type* with ID 4 is node *chapter* with ID 6. Since node *book* with ID 2 is an ancestor of node *chapter* with ID 6, we then can conclude that nodes *author* with ID 3 and *type* with ID 4 are not meaningfully related to each other because node *author* with ID 7, which is of the same type as node *author* with ID 3, is more related to node *type* with ID 4 under the node *chapter* with ID 6, which is actually the MLCA of nodes *author* with ID 7 and *type* with ID 4.

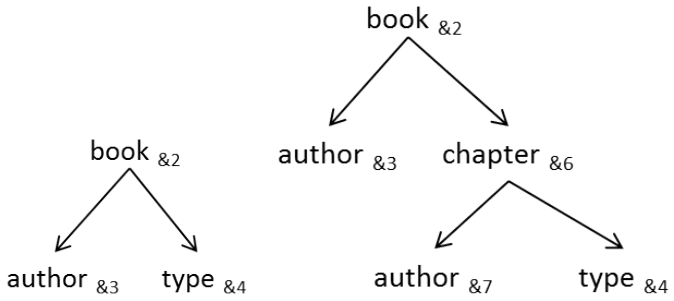


Figure 2.9: Structural relationships among nodes in MLCA heuristic

2.1.6 XML Streams

XML streams is referred to the sequent pieces of XML data that are continuously transmitted in real-time from one location (or devices) to other locations. Different from permanently-stored XML data that the content of the whole XML data is all known and available at the same time, in XML streams' context, only the content of the receiving pieces of XML data is known and ready to be used.

Processing Model of XML Streams

In this part, we describe the system architecture of XML stream processing [25] to illustrate the context in which XPath expressions or keyword search is used.

The users create some queries(XPath or keyword search) and register them to the filtering system. Then an input XML stream is first parsed by a SAX parser that generates a stream of SAX events. Then, these events are input to the query processor that evaluates the queries and generates a stream of application events. These streams of application events are the answers to the registered queries. And these answers are forwarded to the applications or users as shown in Figure 2.10.

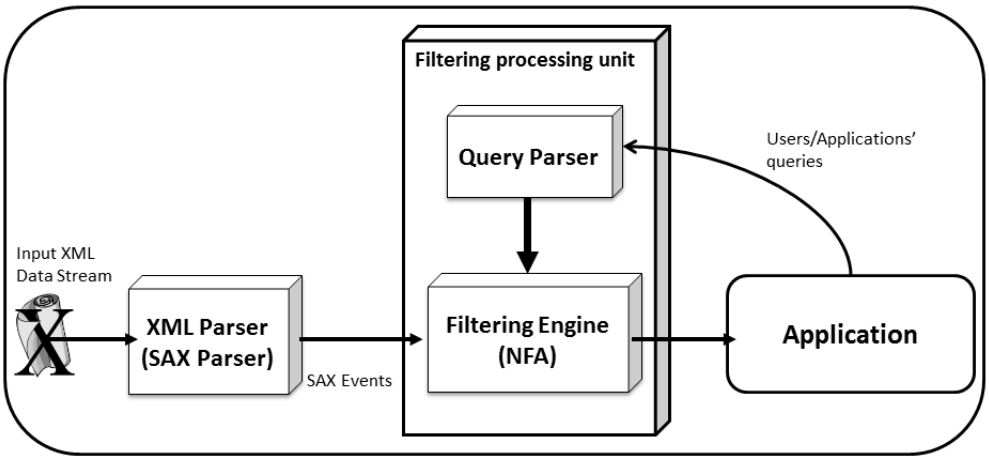


Figure 2.10: System architecture

2.2 Relational Database

Relational database [15] is the digital organization of data that is based on relational model. It was first proposed in 1970 by E. F. Codd of IBM's San Jose Research Laboratory. His proposal to the use of relational database model that database schema is disconnected from physical data storage has changed the way people thought about databases. His proposal has become the standard principle for database systems.

Relational database is the successor of the computerized database that started using in the 1060s when the use of computer became popular and very efficient options for many organizations. In 1960s, several popular computerized databases were CODASYL [16] (a network model), IMS [17] (a hierarchical model), and SABRE system [18] that was used by IBM to help American Airlines manage its reservations data.

Due to its cost-efficient in managing information, relational database has become very popular and has been extensively used in nearly every field, such as science, engineering, technology, business, finances, etc. Here are some popular relational database systems: Oracle [19], MySQL [20], Microsoft SQL Server [21], PostgreSQL [22], and IBM DB2 [23].

2.2.1 Relational Data Model

Database is a collection of interrelated tables (or "relations") that store all data with the same category (same properties) together by using relational model. Each table consists of columns and rows, with a unique key identifying each row. Each table represents one "entity type", row represents each record or tuple, and column represent values attributed to that record or tuple.

Constraints provide ways to implement rule in the database by restricting the data that can be stored in the tables. Two main principle rules are entity integrity and referential integrity. Entity integrity provides mechanism to maintain primary key that are a unique identifier for each row in the table. With referential integrity, foreign key can be maintained. Foreign key is set to the table whose primary key is being referenced. Therefore, tables in the relational model are related with each other by primary/foreign key relationship.

Figure 2.11 is a schema of relational database that consists of three tables, "*Customer*", "*Purchase*", and "*Product*". The primary/foreign key relationship between tables are as shown in the figure. Figure 2.12 is a relational database of the above schema in Figure 2.11. *t1*, *t2*, ..., *t8* marked in red near each table represent tuples. The line linking between tuples represents their relationship.

Figure 2.11: Schema of relational database

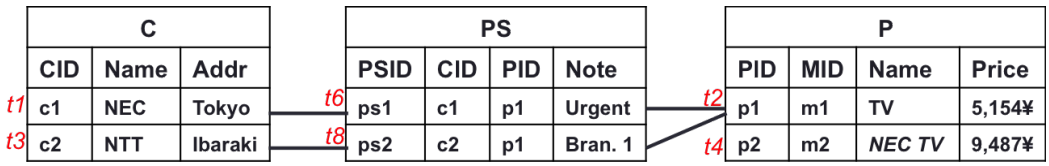


Figure 2.12: Example of relational database

2.2.2 SQL

SQL [24] is one of the first query languages of E. F. Codd's relational data model in 1970. It is a standard query language for accessing and manipulating data from relational database. The scope of SQL includes insert, query, update, and delete. The most common operation in SQL is query. Query uses declarative SELECT statement to retrieve data from tables. Some common clauses in SELECT statement are:

- FROM: Indicate the tables to retrieve data from.
- WHERE: Impose the select condition by restricting the rows returned by the query.
- GROUP BY: Project rows having common values into a smaller set of rows.
- ORDER BY: Define the columns on which the resulting data is sorted. Sorting can be either ascending or descending.

- DISTINCT: Eliminate duplicate rows.

SQL query is difficult to use because it is required user to have knowledge of how to use the query and detail about the schema of relational database, such as fields' name, relationship between tables, etc.

For example, suppose user would like to retrieve all related tuples from tables "*Customer*" and "*Purchase*", following SQL statement is used:

```
SELECT *
FROM Customer, Purchase
WHERE Customer.CID = Purchase.CID;
```

2.2.3 Relational Keyword Search

In this section, we shall first introduce keyword search on relational databases. As a common basis, graph representation of relational database is used to define the semantics of keyword search [54, 58, 62, 71, 75, 75]. In a data graph, each node represents a tuple, and an edge represents a primary/foreign-key reference between two tuples. Now, let us assume a relational schema and a database that conforms to the schema, relational keyword search is defined as follow.

Definition 7. *Relational Keyword Search [54,58,62,71,75,75]: Given a set of user-specified query keywords, $\{k_1, k_2, \ldots, k_n\}$, and relational database, keyword search on the database is to find all possible minimal total joining networks of tuples (MTJNT) [58] that are both:*

- *Total: Every keyword is contained in at least one tuple of the joining network (JNTs).*
- *Minimal: Removing any tuple from a network of tuples leads to loss of eligibility for query results.*

From the definition, all results returned by relational keyword search, which are called MTJNTs, are unique due to the two constraints as defined in the definition. Notice that, MTJNT can contain as many tuples as it can, which makes the number of all results of relational keyword search be very huge. Therefore, the relational keyword search is also given the predefined parameter T_{max} to limit the maximum number of tuples allowed in each MTJNT.

For example, suppose a user would like to search for "*NEC, TV*" from relational database in Figure 2.12. Suppose the maximum number of tuples allowed in the result (T_{max}) is set to five. Some joining networks of tuples (JNTs) corresponding to this keyword search are shown in Figure 2.13. In each JNT, line linking between two tuples represent the relationship between tuples. The answers of this keyword search are joining networks of tuples (JNTs) 1 and 2 as shown in Figure 2.13 because they are MTJNTs; while JNTs 3 and 4 are either not minimal or total.

2.2.4 Candidate Network (CN)

To process relational keyword search over relational database, one typical approach is to find all JNTs that contain all query keywords first, then check if each JNT is an MTJNT

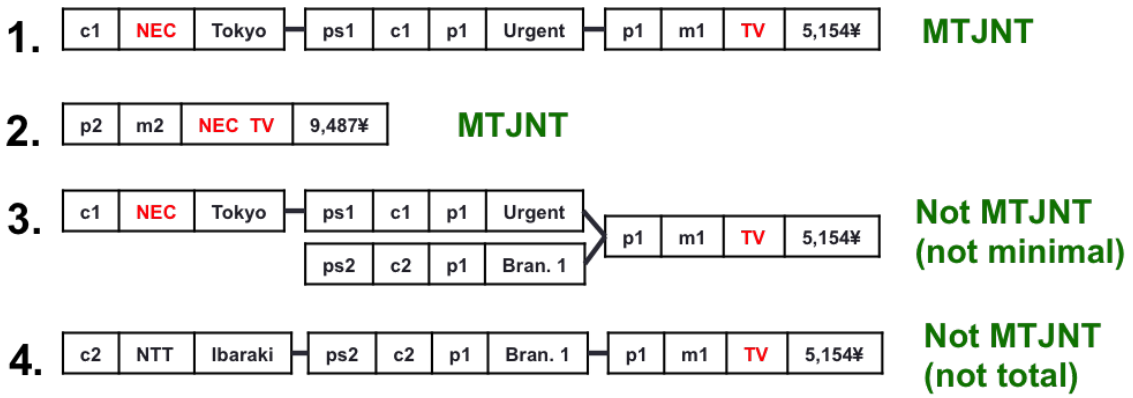


Figure 2.13: Example of MTJNTs for keyword search "*NEC, TV*" on relational database in Figure 2.12

later. This approach is quite inefficient if all matched MTJNTs are needed because each tuple in the relational database needs to be checked a lot of time and every JNT must be redundantly checked if it is MTJNT. That can be done more efficiently if all MTJNTs that are of the same pattern are grouped together. For this purpose, generating all possible matched patterns of all MTJNTs from the given relational keyword search before traversing the tuples in relational database to find matched MTJNTs is a better way. Such matched pattern is called candidate network [58, 75] (CN), which can be generated by using the given query keywords and schema of relational database. Candidate network (CN) is defined as follow.

Definition 8. *Candidate Network (CN) [58, 75]: Given a set of user-specified query keywords, $\{k_1, k_2, \ldots, k_n\}$, and schema of relational database, CN can be generated by eliciting all possible combination of both relational keyword search and schema of relational database. Specifically, all CNs are higher level representation of all MTJTNs at schema level.*

From the definition, since all CNs represent all possible MTJNTs at schema level, there is no redundant CNs. Moreover, each CN must also be both total and minimal. Similarly, the predefined parameter T_{max} is also used to limit the maximum size of each CN. The total number of all CNs can be very huge depending on the schema, number of keywords in the query, and T_{max}. Different from MTJNT that each node represents a tuple, each node in each CN represents a tuple set. The tuple set refers to subset of any table that contains the same keyword(s) or no keyword of the given query.

Figure 2.14 shows all CNs that are generated from schema in Figure 2.11 for relational keyword search "*k1*, *k2*". In each CN in this figure, each node represents a tuple set, and line linked between two nodes represents their relationship. Only one single node (tuple set) is also a CN as long as it contains all keywords of the given query. For example, in CN 1, node (tuple set) "*C*{*k1k2*}" is referred to all tuples from table "*Customer*" that contain both keywords "*k1*" and "*k2*". In CN 5, node "*PS*{}" is referred to all tuples from table "*Purchase*" that does not contain any keyword in relational keyword search "*k1*, *k2*".

2.2.5 Query Plan

Having generated all CNs from the given keyword search and schema of relational database, looking for matched MTJNTs can be done more efficiently by processing all CNs against

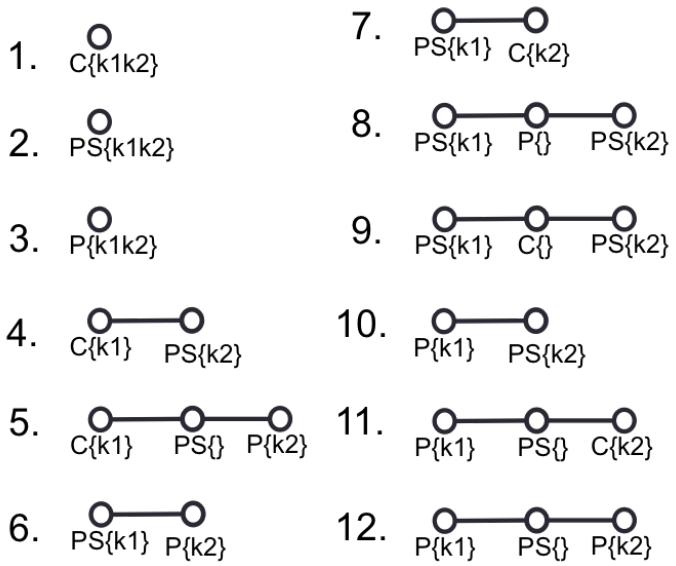


Figure 2.14: All CNs created from schema in Figure 2.11 for query "k_1, k_2". Notice that the label under each node is a tuple set, and "*C*" is referred to table "*Customer*", "*PS*" is referred to table "*Purchase*", and "*P*" is referred to table "*Product*". The keyword inside the curly bracket is referred to the keyword of the given query that each node contains. Notice that, for this example, T_{max} is set to 3.

all tuples in the relational database in stead of directly evaluating the given keyword search. Moreover, among all CNs, some of them might have overlapping part, which can share processing when evaluating tuples from relational database. For example, CNs 11 and 12 in Figure 2.14 have overlapping part, "*P{k1}-PS{}*". Therefore, evaluating these two CNs independently against relational database is not a good option because their overlapping part can share processing. Based on this idea, query plan [58] is created by combining all CNs together in various ways so that processing among CNs can be shared when searching for matched MTJNTs. There are various ways to create query plan. We would like to give a general definition of query plan as follow.

Definition 9. *Query Plan [58]: Given a set $C_1, C_2, ..., C_n$ of CNs, a query plan is created by combining all CNs together in such a way that processing sharing among all CNs when evaluating relational database is maximal:*

- *All matched MTJNTs can be retrieved quickly.*
- *Memory/disk usage is small.*

Figure 2.15 is an example of one possible query plan that is created by combining all CNs in Figure 2.14 together. This query plan is created in such a way that all CNs that have the same nodes containing keyword "*k1*" (black-filtering-circle) are grouped together, called cluster. In each cluster, CNs that have overlapping parts are merged so that processing can be shared. In this example, overlapping part, "*P{k1}-PS{}*" of CNs 11 and 12 is merged in cluster 6th.

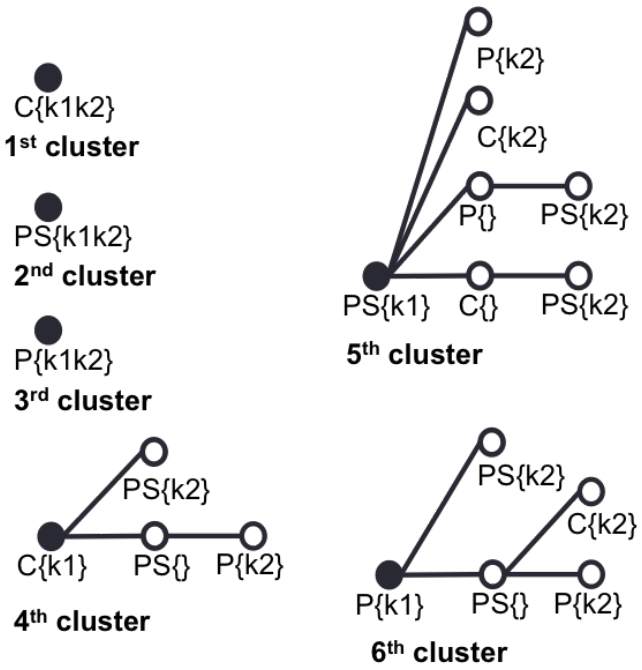


Figure 2.15: Example of query plan created from all CNs in Figure 2.14.

2.2.6 Relational Streams

In contrast to conventional relational data, *relational streams* [55] can be modeled as possibly unbounded sequences of relational tuples that conform to relational schemas. In other words, each tuple in a stream can be represented by a pair of 1) a relational tuple, and 2) a time instant of a discrete and ordered time domain, e.g., integer. Thus tuples are regarded that they are arrived according to their timestamps. Figure 2.16 illustrates a sample relational streams of schema in Figure 2.11. Column "*Time*" defines the timestamp of when tuple arrives from streams.

When dealing with (relational) streams, we often use *sliding windows* to convert an infinite stream of tuples to a relation of finite tuples. In such window semantics, two tuples can be joined only if both tuples are in the sliding window.

Processing Model of Relational Streams

Having defined relational streams and sliding windows, keyword search over relational streams can be defined as follow:

Definition 10. *Keyword Search over Relational Streams: Given a set of query keywords,* $\{k_1, k_2, \ldots, k_n\}$*, a maximum network size* T_{max}*, and a window specification W, it continuously:*

- *Reports new MTJNTs when new tuples are delivered.*
- *Invalidate existing MTJNTs due to deletion or aging of tuples.*

Figure 2.17 shows system architecture of keyword search over relational streams, which comprises of two main steps: *preprocessing* and *filtering* steps.

	C					PS						P				
	CID	Name	Addr	Time		PSID	CID	PID	Note	Time		PID	MID	Name	Price	Time
t1	c1	NEC	Tokyo	1	t6	ps1	c1	p1	Urgent	6	t2	p1	m1	TV	5,154¥	2
t3	c2	NTT	Ibaraki	3	t8	ps2	c2	p1	Bran. 1	8	t4	p2	m2	*NEC TV*	9,487¥	4

Figure 2.16: Example of relational streams of schema in Figure 2.11.

Preprocessing step Given a schema, a set of query keywords, and T_{max}, all *Candidate Networks* (CNs) [59, 61] are generated. Then a query plan is generated from all CNs.

Filtering step In this step, the query plan is evaluated over relational streams. When new MTJNTs are detected due to arrivals of new tuples, they are reported. On the other hand, expired tuples are removed by using either eager or lazy approaches [59].

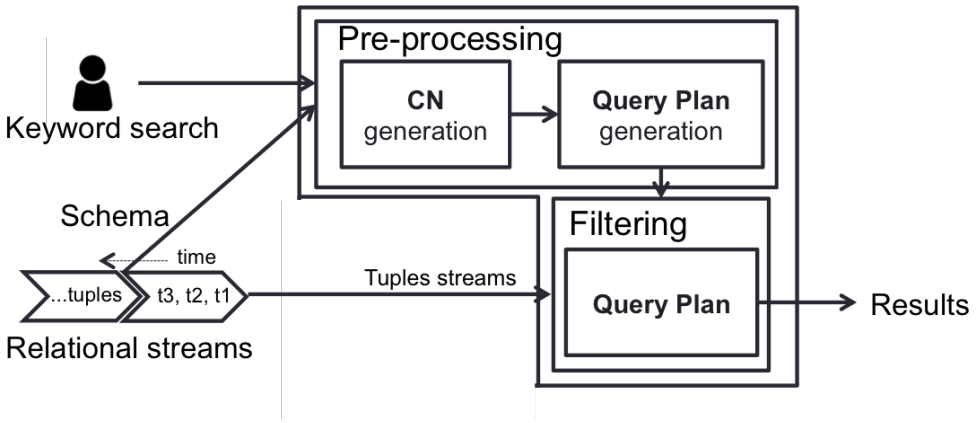


Figure 2.17: General framework

Chapter 3

Survey and Related Works

3.1 Survey

Looking for interesting information from data streams in real-time is quite useful. That can be done by using queries. Queries used to extract information from data streams are different according to the targeted data streams. Typically, they can be categorized into two groups: 1) structured query, and 2) keyword search. Structured query is referred to the standard query designed specifically for one particular data streams. For example, SQL/CQL are exclusively used in relational data/streams, and XPath expression is for XML/XML streams. To use such structured queries, knowledge about how to use queries and structural information of data sources are required. keyword search is a user-friendly-search technique. It is not designed specifically for any specific data streams, and can be used for any data streams. Moreover, using keyword search requires neither knowledge of structural information of data streams nor knowledge of query itself. The survey regarding query processing of data streams is presented in Table 3.1 and is explained in the following sections.

3.1.1 Query Processing on Structured Data Streams

Structured Query

Structured query processing over data streams has been extensively studied. Searching for subgraph from graph streams has been studied in [94–96]. In these studies, given a query graph, various algorithms were proposed to efficiently retrieve all subgraphs that approximately matched the query in real-time. Correlation coefficients were developed to compare the difference between all subgraphs from streams with the given query graph. If the resulted correlation coefficient was greater than a given threshold, the respective subgraphs were qualified to be the query's results.

Query processing of relational streams [89–93] has been a hot research topic for a long time. So far, there are a lot of research works and products of relational streams that have been developed and extensively used in various fields in real life. In these works, an SQL query for relational streams, called CQL, was developed and used to retrieve important information in real-time. CQL can be very complicated, and together with the unpredictable nature of relational streams, so to efficiently process CQL is very challenging. This problem got more complicated when it was required to process large number of CQLs at the same

Table 3.1: Survey on query processing of data streams

Type of data streams	Types of queries: Structured query	Types of queries: Keyword search
Structured data	• Graph query on graph streams [94–96] • CQL query on relational streams [89–93]	• Keyword search on relational streams [59–61]: • S-KWS [59] • SS-KWS [60, 61]
Semi-structured data	• JSQ on JSON streams [106] • C-SPARQL/CQELS on RDF streams [107–111] • XQuery/Twig query on XML streams [112–120] • XPath on XML streams [25, 35, 37, 41, 44, 49, 51, 52]: • YFilter [35]	• Keyword search on XML streams [26, 27, 99–105]: • CKStream [26]
Unstructured data		• Keyword search over text streams [127, 128] • Pub/sub over dynamic event streams [122–124] • Etc.,

time. To efficiently process multiple CQLs, query plan was typically used by combining all CQLs together so that all queries could share processing as much as possible. Most works on processing relational streams predetermined the order of processing of each component in the query plan (e.g., component that was shared by most queries was to be processed first, and the less shared component was to be processed the latest). Such presumption sometimes made the processing of the query plan worse if actual data from relational streams did not favor the plan. Recently, the work of pattern sharing of pattern queries on event streams [121] addressed the above problem by proposing a dynamic query plan for pattern queries. In that work, the order that each component in the query plan was processed was changed dynamically as data came from relational streams. The main idea was that the relational streams that had been processed so far was kept and used to compute the cost of processing query plan by using the existing order and cost by using other potential possible orders. Then, the costs were compared, and if the cost of processing query plan by using the new order was better than the existing one, the new order would be used. Even though some performance improvement could be achieved, periodically computing cost of executing the existing query plan and migrating existing query plan to adopt the better processing order could be very costly.

Keyword Search

There have been extensive studies for processing of structured queries over structured data streams; however, to the best of our knowledge, there have been only two works that have

focused on processing keyword search over structured data streams. The structured data streams that has been focused on is relational streams. A survey [150] by Xu et al. (2010) pointed out that there were a lot of works done on processing of keyword search over static relational database, but there were only two works focusing on keyword search over relational streams. They are S-KWS [59] and SS-KWS [60, 61].

The pioneering work for this search framework was proposed by Alexander et al. (2007) [59]. In this work, the schema of relational streams was assumed to be known in advance. In stead of directly processing the given keyword search against tuples that continuously arrived from streams, all possible matched patterns were pre-generated by using the set of keywords in the given queries and schema of relational streams. Such matched patterns were called candidate network (CN). Then, all CNs were combined into one query plan, which was called operator mesh, so that all CNs could share processing as much as possible. Then, the query plan was directly evaluated against tuples that continuously arrived from streams to find the matched results. There were two main bottlenecks of this query plan. First, every partial results were kept throughout the entire processing which required extremely large memory usage. Second, there were a lot of CNs that could not share processing in this query plan, therefore its filtering performance was very poor. The succeeding work [60, 61] proposed an alternative query plan, called lattice, in an attempt to achieve better performance improvement by addressing the above bottlenecks. Though, significant performance improvement could be achieved by using lattice, still its performance to filter relational streams was unreasonably bad and not appropriate in real world scenario when processing longer queries (queries have more number of keywords).

3.1.2 Query Processing on Semi-structured Data Streams

Structured Query

Processing structured queries on semi-structured data streams has been extensively studied before, such as JSQ queries on JSON streams [106], C-SPARQL processing on RDF streams [107–111], and XPath/XQuery/twig queries on XML streams [25, 35, 37, 41, 44, 49, 51, 52, 112–120].

JSON (JavaScript Object Notation) has gaining more attention because of its lightweight data-interchange format, and more information is being exchanged in real-time in the form of JSON. Abakumov et at. (2013) [106] proposed a distributed query for JSON stream, called JSQ (JSON Stream Query) by addressing all features of this query language. However, no discussion about how JSQ could be efficiently processed against JSON streams.

The very first SPARQL engine for dealing with RDF streams was proposed in [110]. This engine was built based on two main principles: 1) prune unrelated input RDF streams as soon as possible, and thus saved processing cost in terms of both space and time, and 2) determine the results of the queries as soon as possible. Barbieri et al. (2010) [108] formally proposed C-SPARQL as a formal query language for RDF streams by additionally introducing new features relative to SPARQL. Another alternative query was CQELS [111]. The processing of C-SPARQL/CQELS was similar to that of CQL over relational streams by decomposing all queries into parts and determining which part to be processed first for good efficiency. Moreover, most existing techniques for this search framework were based on the triple stream model, where each element within a stream was a triple (subject, predicate,

object). DIONYSUS [109] has recently proposed by indicating that existing RDF streams processing systems could be categorized into three groups: 1) batch processing of the distributed RDF streams that were optimized from static data processing, 2) centralized RDF streams systems that only provided real-time analytics, and 3) RDF streams systems that only provided sequence operators. The main aim of DIONYSUS was to build a general-purpose system that consisted of all the capabilities of all systems of the three categories above. Though, developing such general-purpose system was difficult, and in DIONYSUS, there was no concrete idea and algorithm being discussed. Even though there had been a great deal of efforts put into processing of RDF streams, achieving good performance in terms of filtering throughputs was still the main problem when the input queries gets longer or more complicated.

Due to the popularity of XML, the studies of the processing of XML streams [25, 35, 37, 41, 44, 49, 51, 52, 112–120] have been extensively done for a long time. The popular queries are XQuery [112–117], twig query [118–120], and XPath expression [25, 35, 37, 41, 44, 49, 51, 52]. XQuery and twig query were more expressive than XPath expression in querying XML streams, but they were much more complicated and processing is much more costly. Using these queries required users to know how to use the queries themselves and the detail information about schema of XML streams. Since the processing of XQuery and twig query was similar to that of structured query on structured data streams, we would only review the processing of XPath expression over XML streams as follow. XPath processing over XML streams could be categorized into three groups: 1) prefix-sharing approach, 2) postfix-sharing approach, and 3) prefix-and-postfix-sharing approach. We would like to review research activities of each approach as follow.

[35] developed YFilter, an XML filtering system aiming at providing efficient filtering for large number of XPath queries. Different from previous works on processing XPath over XML streams, YFilter employed the Nondeterministic Finite Automaton (NFA) that combined all queries into a single machine. That NFA was one of the key improvements of YFilter. The commonality among XPath queries were well studied and they found out that, by merging the common prefixes of the XPath queries, the machine states became very compact. As a result, the shared processing mechanism provided great improvement in structure matching performance over algorithms that did not exploit such shared processing or only exploited very limited extent of XPath sharing. In addition to the above significant innovations, the relatively small number of machine states, incremental machine construction, and ease of maintenance were the remarkable characteristics of the NFA-based implementation of YFilter. In order to deal with multiple XML schemas, two approaches were very common: to apply query rewriting [37] and to use global schema [36]. [25] proposed a lazy DFA-based filtering system which was superior to the NFA-based filtering system in terms of processing performance. In the NFA-based filtering system, the throughputs were constantly decreasing as the number of XPaths increases. This problem could be solved in the lazy DFA-based filtering system, because it ensured constant high throughputs regardless of the increase in the number of XPaths. However, there were several drawbacks of the DFA-based filtering system, one of which was its excessive consumption of memory caused by a large number of DFA states, and thus the system could run out of the memory.

PostFilter was proposed in [52] to address the problem of XML filtering that exploited the prefix commonalities among path expressions. Such prefix-path-sharing systems suffered from the explosion of NFA states when XPath expressions contained ancestor-descendant

axis ("//"). Such an explosion of NFA states could be solved by exploiting postfix sharing among XPath expressions. If XPath expressions that began with ancestor-descendant axis ("//") were used often, such queries were most likely to have the postfix sharing. Therefore, to support postfix sharing capability, a bottom up filtering approach exploiting postfix sharing was proposed. They proved with experiment that PostFilter works better than other prefix-sharing filtering systems if the input XPaths contained more ancestor-descendant axis ("//").

AFilter [53] was proposed to take advantage of both prfix and postfix sharings to reduce the overall filtering time and increase throughputs. In [52], only postfix commonality was used; while [35] only used prefix sharing among the input XPaths. Different from previous works, AFilter made use of the capability of both prefix and postfix commonalities among XPath expressions. It was not an automata-based approach. It introduced its new memory organization and structure matching. AFilter provided balance between memory usage and performance speed up. When the allocated memory was exhausted, additional memory could be exploited. Its memory supply was flexible because it used an on-demand prefix caching mechanism (PRCache). Its memory organization consisted of common steps (AxisView), common prefixes (PRLabel-tree), and common suffixes (SFLabel-tree). Prefix and suffix labels were generated by the PRLabel-tree and SFLabel-tree respectively. Those labels were the main structures of AxisView edges. As a consequence, exploiting both prefix sharing and suffix sharing simultaneously lead to higher performance than was achieved by relying on only one option.

Keyword Search

There are very few works done to enable keyword search over semi-structured data, and all of them focus on keyword search over XML streams. To the best of our knowledge, there are only four works done to enable the processing of keyword search over XML streams. They are SKStream [27], CKStream [26], MKStream [101], and PMKStream [100]. We would like to review them as follow.

The works in [26, 27] took the first step towards processing keyword search over XML streams. In SKStream [27], they introduced sophisticated query processing algorithms that could answer keyword search over XML streams. This work was more novel than the previous works, which mainly worked on static XML data. The used node relatedness heuristic was the famous SLCA (Smallest Lowest Common Ancestor) heuristic [34]. Moreover, they used a stack to store all nodes that were processed. Each processed node was associated with a bitmap, where each of its bits is associated with each query term in the query. However, this work only supported a single keyword search over XML streams. To fulfill this incompleteness, the work in [26] proposed multiple keyword searches over XML streams. They proposed two new algorithms, KStream and CKStream, for simultaneously processing several keyword searches over XML streams. They relied on parsing stack and query index specially designed to allow the simultaneous matching of the terms from different queries. Similar to [27], this work used SLCA heuristic [34] to answer the keyword search. Later one, MKStream [101] was proposed in an attempt to make performance improvement over CKStream by proposing to use multiple stacks. The main idea of this work was to divide queries into groups so that queries in each group could share the most keywords. Queries that were in the same group were simultaneously evaluated together by using one independent stack. Therefore, the total number of used stacks were equal to the number of queries'

groups. The most recent work for this search framework is PMKStream [100]. This work was proposed so that MKStream could be processed in parallel. Performance improvement could be significantly achieved. Notice that, all the works above adopted the concept of SLCA (Smallest Lowest Common Ancestor) [34] to judge which matched sub-trees were qualified to be queries' results. Though, SLCA was not effective enough to prune the results that were not related with users' search intentions. Therefore, the accuracy of search results of these approaches was still the main problem. It is noted that MKStream and PMKStream were proposed after the first proposed approach of this dissertation was done.

3.1.3 Query Processing on Unstructured Data Streams

Keyword Search

Based on our survey, there is no work that has been done to enable the processing of structured queries over unstructured data streams, but processing keyword search over unstructured data streams has been extensively studied [122–124, 127, 128].

The works [127, 128] studied the problem of answering keyword search on multiple text streams. A result to a query was a text or a set of correlated texts that contained all query keywords within a specified time span. The main idea of this approach was to keep every text streams within the specified time span and eagerly looked for the sets of correlated texts that contained all query keywords every time new text streams arrived. Finally, if any set of correlated text streams was found, it was checked if it was a qualified result. Notice that, in this work, the correlation between two texts was determined by the defined model. The main concern of this approach was that every time new text streams arrive, checking must be done against all existing valid texts seen so far. Therefore, there were a lot of redundant checkings, and that could not guarantee that there were matched results.

[122–124] investigated the problem of efficiently support location-aware publish/subscribe system. In such system, subscribers registered their interests as spatial-keyword subscriptions. When any incoming geo-textual message matched the registered subscriptions, the respective subscribers was alerted with the matched message. In stead of using existing query indexes that always pruned the incoming message by spatial constraint before keyword constraint, [123] proposed AP-tree that was builded so that order of checking spatial constraint and keyword constraint were done dynamically. The main idea is that if checking spatial constraint first could lead to prune more number of unrelated subscriptions than doing so on keyword constraint, then checking spatial constraint was done first. Otherwise, keyword constraint was checked first. However, this AP-tree only supported non-moving spatial-keyword search over streams of geo-textual messages. In many real world scenario, it was very typically that published/subscribed systems supported moving spatial-keyword search over streams. For this purpose, [123] was done to extend AP-tree so that it supported moving queries. The new extending index was called AP$^+$-tree. Another work that has been done to enable the processing of moving-spatial keyword search over streams is Elaps [124]. This work adopted the safe region and impact region for each subscriber to minimize the communication overheads when subscribers moved or new spatial-textual messages arrived. For example, if subscribers moved within their safe region, it could guarantee that there was no neither new matched spatial-textual messages nor their existing matched spatial-textual messages became unmatched. Similarly, when new incoming spatial-textual messages ar-

rived and did not fall into the impact regions of any subscribers, it could guarantee that there was no new match. However, computing safe region and impact region must be done very frequently and could be very costly. Notice that the papers of AP$^+$-tree [123] and Elaps [124] were published at the same year, so there was no performance comparison between them.

3.1.4 Summary of the Survey and Position of this Dissertation

This chapter briefly surveys all main researches about query processing over data streams. Existing algorithms are categorized according to types of data streams: structured query and keyword search over structured data streams, semi-structured data streams, and unstructured data streams. As explained above, there are extensive studies of the processing of structured queries over structured and semi-structured data streams. However, there are very few studies on the processing of keyword search over structured and semi-structured data streams. This dissertation focuses on keyword search over structured and semi-structured data streams. Our brief survey shows that the existing algorithms still suffer from two main performance bottlenecks.

One critical bottleneck is the very poor performance of the existing approaches in terms of filtering speed when processing longer queries. The reason is that when processing queries that have more number of keywords, the number of partial results (candidate results), which need to be kept and instantly evaluated with the future incoming streams, are exponentially increased. The existing algorithms cannot handle that well by letting a lot of partial results be independently evaluated against the future incoming streams, which causes the performance degrade so much that is not suitable for real search engine. Same problem happens when the maximum size of the search results are set to be big (T_{max}). As explained in Section 1, these two parameters (long query and big T_{max}) are very important for real search scenario.

Another problem of the existing algorithms is how to efficiently return only the results that are really needed by the users. The existing algorithms of keyword search over structured and semi-structured data streams return all results to the users even though they are not what users want. This problem happens because the existing algorithms do not understand the real search intentions of users through queries that consist of sets of pure keywords. Specifying real search intension with just pure keyword search is difficult because keywords can appear in any parts of data streams and can carry multiple meanings. This problem can be partly solved by adopting the ranking mechanisms [28, 29, 31, 32, 43, 58, 62–65] that have been extensively studied for keyword search over static data. However, such rankings are sometimes not effective because they exclusively use the whole static data and query (a set of pure keywords) to rank the results without taking into account what the users really want to get. To the best of our knowledge, existing algorithms of keyword search over structured and semi-structured data streams do not adopt such ranking mechanisms. Nevertheless, we believe that such ranking mechanisms can be adopted to streams' setting, though they may become less effective and involve in very heavy computational cost due to the unavailability of whole data streams at the time of ranking. That will put additional burden to the already poor performance in terms of filtering speed as explained above.

As part of the solutions to the above two critical problems, this dissertation presents two efficient algorithms for each problem. With regard to structured and semi-structured data streams, we address the first problem by focusing on keyword search over relational streams, and the second problem by focusing on keyword search over XML streams.

Table 3.2: Related works and position of this dissertation

<table>
<tr><th rowspan="2">Type of data streams</th><th colspan="2">Types of queries</th></tr>
<tr><th>Structured query</th><th>Keyword search</th></tr>
<tr><td>Structured data</td><td>• Graph query on graph streams [94–96]
• CQL query on relational streams [89–93]</td><td>• Keyword search on relational streams [59–61]:
• S-KWS [59]
• SS-KWS [60, 61]
• Improved keyword search (2nd proposed work in Chapter 5.)</td></tr>
<tr><td rowspan="2">Semi-structured data</td><td>• JSQ on JSON streams [106]
• C-SPARQL/CQELS on RDF streams [107–111]
• XQuery/Twig query on XML streams [112–120]
• XPath on XML streams [25, 35, 37, 41, 44, 49, 51–53]:
• YFilter [35]</td><td>• Keyword search on XML streams [26, 27, 99–105]:
• CKStream [26]</td></tr>
<tr><td colspan="2">XPath-based keyword search (1st proposed work in Chapter 4.)</td></tr>
<tr><td>Unstructured data</td><td></td><td>• Keyword search over text streams [127, 128]
• Pub/sub over dynamic event streams [122–124]
• Etc.,</td></tr>
</table>

For the first problem, which is specifically about relational streams, our algorithm proposes an efficient query plan that can handle all candidate results more effectively against the future incoming relational streams. This is done by keeping all candidate results together according the query plan in such a way that evaluating them against future incoming relational streams can be done as minimal as possible, which means the number of candidate results that can share processing is maximal. Therefore, longer queries can be handled well. This proposed algorithm is explained in Chapter 5.

For the second problem, in stead of adopting the ranking mechanisms [28, 29, 31, 32] of static XML, which is sometimes not efficient and too costly for XML streams, we propose a user-friendly query that allows users to easily and effectively define their real search intentions. For this purpose, we propose XPath-based keyword search that combines XPath with keywords. Specifically, XPath- part is used to specify which part of XML streams that users want to search, and keywords- part is used to define search intention for the query results. To the best of our knowledge, there is no existing algorithms that have been done to process this kind of query over XML streams yet. This contribution is presented in Chapter 4.

The position of this dissertation with respect to the above survey is shown in Table 3.2. Notice that the parts written in clear black are works related to this dissertation (parts in pale are not much related).

3.2 Related Works

In this section, we would like to review in detail the works that are highly related to the proposals of this dissertation as shown in Table 3.2. These works are the comparative approaches that will be used to compare the performance with the proposed approaches. We divide these comparative approaches into two for each proposal: 1) related works for XPath and keyword search over XML streams, and 2) related works for keyword search over relational streams.

3.2.1 Related Works for XPath and Keyword Search over XML Streams.

First, we would like to review the comparative approaches of the first proposal, XPath-based keyword search over XML streams. Since this proposed approach is related to both XPath search and keyword search over XML streams, we would like to review both approaches by using YFilter and CKStream. The reason that we choose YFilter rather than PostFilter and AFilter because our aim for the first proposal is to find XML fragments that users want to search before finding sub-trees in those fragments that satisfy keyword specification. For this purpose, technically, prefix-sharing-based approach of YFilter is more appropriated. Moreover, due to the nature of XML and XML streams that can always be viewed as trees in hierarchical form from root to leaf, there are most of the time, though not always, more common prefixes than postfixes among all sub-trees. Therefore, we believe that YFilter is more standardized as an efficient approach than PostFilter and AFilter. Regarding choosing CKStream rather than MKStream and PMKStreams is that, at the time of doing research of the first proposed approach, CKStream was the newest approach. Though, we believe that our first proposed approach can be easily adopted the methods used in MKStream and PMKStreams.

Keyword Search over XML Streams (CKStream)

Keyword search over XML streams is a searching technique where the inputs are a set of queries that contain keywords and XML streams, and the outputs are XML sub-trees that contain all keywords. Specifically, the input queries are evaluated against XML streams where XML elements and their textual values continuously arrive. During the entire filtering, all sub-trees of XML streams are kept, and when any sub-tree is detected as containing all keywords of any query, such sub-tree is returned as the query's result.

However, given a set of queries (keyword search), processing them independently against XML streams is not efficient because different queries might have common keywords. For this purpose, CKStream [26] was proposed. Specifically, given a set of keywords, CKStream creates a compact query index storing all unique keywords, which is used to evaluate against XML streams. Each index entry represents a query term and refers to queries in which this term occurs. After query index is built, it is used to evaluate against XML streams. To track matching status of sub-trees of XML streams, CKStream uses query bitmap. The size of query bitmap is the same as the total number of unique keywords of all input queries, therefore each of its bit is associated with one unique search term. Notice that, CKStream uses parsing stack to keep track of all sub-trees together with their matched statuses (query bitmap).

A sample query index created from queries q_1, q_2, and q_3 above (the samples queries

used in Section 2.1.4) is shown in Figure 3.1. Notice that, query index differentiates between structural (label) and non-structural (value) query terms. Keywords on index 1, 2, and 4 are in the form *l::k*, which requires to be matched on both structural part and non-structural part. Whereas keywords on index 3 and 5 are in the form *k*, which matching can be on either structural part or non-structural part. A sample query bitmap from queries q_1, q_2, and q_3 above is shown in Figure 3.2.

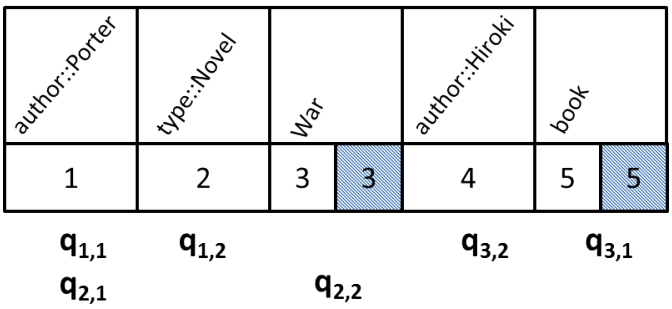


Figure 3.1: Example of query index

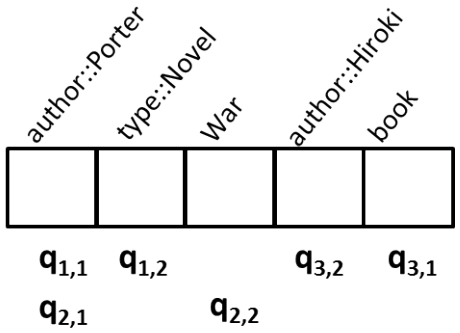


Figure 3.2: An example of query bitmap

XPath Search over XML Streams (YFilter)

XPath search over XML streams is a searching technique that process XPath queries over XML streams where XML elements and their textual values continuously arrive to find matched XML nodes or XML sub-trees.

YFilter [35] is a famous search engine for this search framework 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 Automaton (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.

Figure 3.3 shows the NFA fragments of the basic location steps. There is a transition from one state to another state via a directed edge representing a transition. The symbol

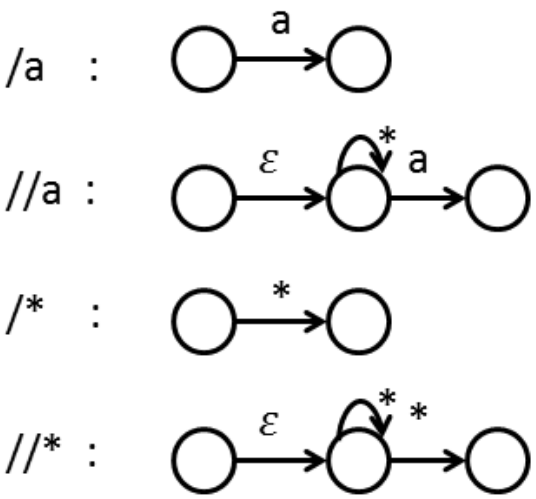


Figure 3.3: Basic NFA location step

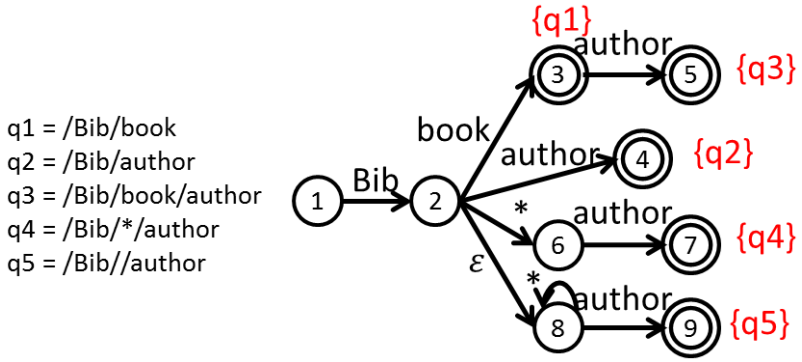


Figure 3.4: XPath queries and a corresponding NFA

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 3.4 shows an example of such a Non-deterministic Finite Automaton (NFA) corresponding to five XPath queries. A circle denotes a state. Circles with double lines denote the accepting states, marked by the IDs of the accepted queries.

When processing an XML data, it is parsed with a SAX parser [14], which is an event-based XML parser; whenever it reads new XML constructs, such as start- and end-tags, text contents, etc., it raises corresponding events and notify to the application program. When YFilter receives a start element event, it triggers a state transition in the Finite State Machines (FSM), while an end element event is received, YFilter must backtrack to the previous states. A run-time stack is used to track the active and previously processed states.

Figure 3.5 shows a running example of the run-time stack, which processes queries shown in Figure 3.4 by XML data shown in Figure 2.1. The content of stack is a set of the active states' IDs. 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 the 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

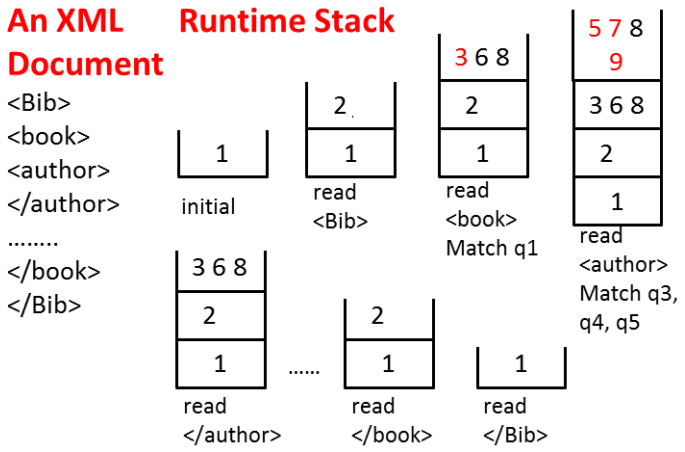


Figure 3.5: An example of query processing in YFilter.

according to these same rules.

3.2.2 Related Works for Keyword Search over Relational Streams

Next, we would like to review the comparative approaches to the second proposal, keyword search over relational streams.

Keyword search over relational streams is a user-friendly search technique where the input is a set of keywords and outputs are tuples or sets of connected tuples that contain all query's keywords. The main idea of this search framework is to continuously evaluate all tuples that continuously arrive from relational streams against all query's keywords. During the entire processing, all valid tuples and partial results (all sets of connected tuples that contain some query's keywords) are kept for evaluating against the future incoming tuples. When any tuple or set of connected tuples is detected as containing all query's keywords, it is regarded as a query's result.

However, keeping all partial results is very costly and requires extremely large memory. To address this problem, S-KWS [59] and SS-KWS [61] were proposed. The main idea of these approaches is that, in stead of directly evaluating all keywords against relational streams, they generate all possible matched patterns called candidate networks (CNs) by using the given keyword search and the schema of relational streams, and then use them to evaluate against relational streams. By using such matched patterns or CNs, some partial results can be reduced, and some performance improvement can be achieved.

In addition, independently evaluating each CN is not efficient because some CNs might have overlapping edges that can share processing. For this purpose, S-KWS [59] and SS-KWS [61] proposed different query plans for efficient processing. The query plans of both approaches are explained in the following sections.

S-KWS

S-KWS [59] is one of the pioneering works for this search framework. In this work, for each CN, the *root* node is defined as the node containing one chosen query keyword. Then, left-deep operator tree is created for each CN.

To improve performance, they propose a query plan, called operator mesh, by grouping all left-deep operator trees that share the same root into a cluster so that all common join operators can be consolidated, resulting in improved performance by sharing common operations on the same data. For example, suppose we have keyword search, $\{k_1, k_2\}$, over relational streams whose schema is shown in Figure 2.11, and T_{max} is set to three, all CNs that can be generated are shown in Figure 2.14. Suppose in this case, root node is chosen as node that contain query keyword, k_1. Then, operator mesh is created by combining all CNs into different clusters as explained above as shown in Figure 3.6.

When processing relational streams, all partial results are cached in each operator's buffer for efficient retrieval of matched results. However, caching all partial results is the main performance bottleneck due to its high memory cost.

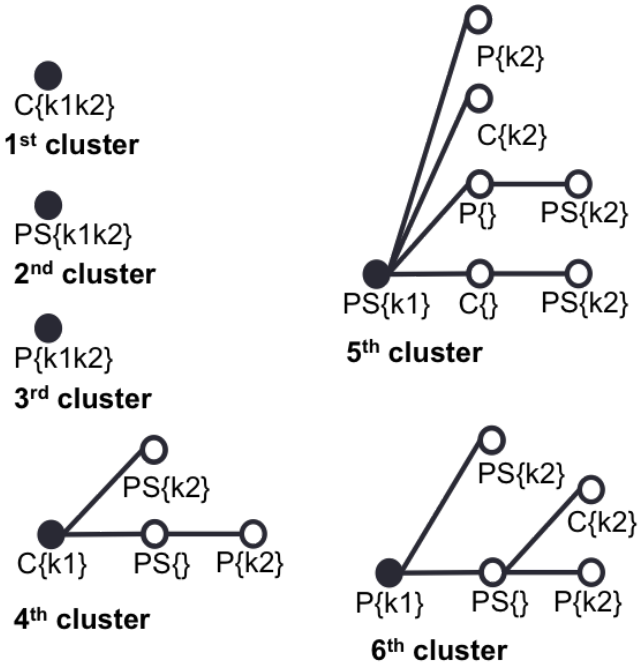


Figure 3.6: Operator mesh that have several clusters created from all CNs in Figure 2.14. Notice that black-filtered circles are root nodes.

SS-KWS

SS-KWS [61] is a successor of S-KWS and can be regarded as the state-of-the-art approach. The novel idea of SS-KWS is to aggressively merge more sub-networks in CNs not only at a single leaf, but also at all leaves. Unlike S-KWS, the root is the center node of the CNs. Besides, instead of operator mesh, a query plan, called lattice, is created by combining all CNs so that the query processing cost is reduced by sharing common subtrees except for the root nodes in CNs as much as possible. Therefore, if T_{max} is set to smaller than four, lattice will be all CNs that are disconnected from each other. With the same example above that T_{max} is set to three, lattice for all CNs in Figure 2.14 is shown in Figure 3.7. As can bee seen, the resulted lattice consists of all disconnected CNs. In this example, nodes marked with double lines are root nodes; black colored nodes are leaf nodes; and the rests are other non-leaf nodes. For node $C\{k_1k_2\}$ acts as both root and leaf node.

To fully reduce all partial results, SS-KWS proposes selection/semi-join approach by dividing buffer of each node into three sub-buffers: N (not joinable), W (waiting), and R (ready). It adopts a bottom-up probing sequence. If the tuple is joinable with other tuples, it is stored in sub-buffer W; otherwise, in N. If MTJNT of any CN is detected, all related tuples are stored in sub-buffer R. Thus SS-KWS successfully reduces memory usage compared to S-KWS.

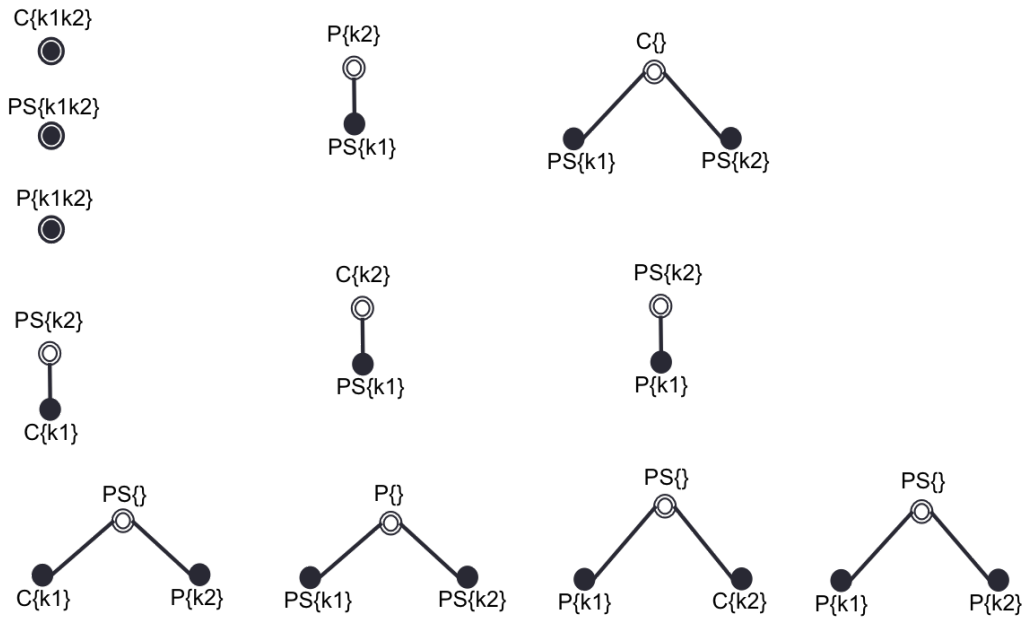


Figure 3.7: Lattice for all CNs in Figure 2.14

Scalability Issues in Existing Approaches

We discuss in detail the scalability issues of these approaches. As a common problem, the number of CNs grows exponentially as the number of keywords and/or T_{max} increase. This gives a significant impact on both time and space.

In S-KWS, partial results are maintained in the buffers in an operator mesh. Due to the low sharing rate of the common subtrees in CNs; e.g., in the operator mesh, we can find a lot of edges connecting the same tables but are not consolidated, because they are either in different clusters or do not have the same root node. Consequently, in query processing, a lot of partial results have to be duplicated in buffers and need to be processed independently.

The problem of the low sharing rate of the common subtrees also happens in SS-KWS because of the restriction that it is impossible to consolidate common paths in the internal nodes because 1) sharing is only allowed for common subtrees, and 2) root nodes are not allowed to be shared. Therefore, the number of unconsolidated paths grows rapidly as the number of CNs grows. For the same reason as discussed above, such duplicated paths cause high memory consumption in the internal buffers and also cause high computational cost for possibly useless processing of (duplicated) intermediate results.

Chapter 4

XPath-based Keyword Search over XML Streams

4.1 Proposed Scheme

4.1.1 Proposal Overview

We propose an efficient algorithm to process XPath-based keyword search over XML streams. For this purpose, we extend the NFA location steps of YFilter that only supports XPath query so that it supports XPath-based keyword search too. Query bitmap of CKStream is adopted and used along with this proposed extended-NFA location steps for efficient tracking of the queries' matching statuses. By using this extended-NFA location step, all given XPath-based keyword searches are merged by using prefix-sharing framework to create a single extended-NFA for efficient query processing. When new XML constructs continuously arrive from XML streams, they are used to evaluate over the single extended-NFA by moving from initial/current states to subsequent states. During the entire processing, all XML sub-trees inside the defined search boundaries, which are expressed by the *XPath-* part of XPath-based keyword search, are kept. When accepting states are reached, the query bitmaps of the corresponding sub-trees are updated. If the query bitmap satisfies any query, the matched sub-tree is returned as a search result. Notice that both SLCA [34, 151] and MLCA [36] can be easily adopted to our proposed algorithm. In this paper, we use both SLCA and MLCA in different experiments.

4.1.2 Combining XPath with Keyword Search

We combine XPath with keyword search by using keywords to specify a query predicate in an XPath expression. In fact, XPath Full Text 1.0 [30] is a W3C standard for that purpose, but its syntax is very complicated. It requires users to have complete knowledge of the syntax of XPath Full Text and to know the detail of XML structure to issue XPath Full Text. Moreover, it not applicable in streams framework. Since our objective is to combine keyword search with XPath-based query, we partially borrow the syntax from it. The resulted syntax is much more simplified and user-friendlier. Here is the basic syntax.

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

where the part "*/XPath-*" is an XPath expression, "*keyword-search query*" contains a set of keywords, and *ftcontains* is a dedicated function to specify a keyword search query according to [26, 27, 29]. Note that the part "*/XPath-*" and "*keyword-search query*" are connected by a descendant axis.

However, one might argue that issuing the proposed XPath-based keyword search is not user-friendly enough because to formulate the part "*/XPath-*" also requires knowledge of the XPath expression and XML structure. However, the part "*/XPath-*" in our query is much more simplified than the traditional XPath expression, and issuing it requires very little knowledge of the XPath expression and does not require any knowledge of XML structure. The only knowledge that users need is how the descendant axis, "*//*", or the child axis, "*/*", can be combined with one of the to-be-searched keywords, which is known by the users as the XML element specifying the XML fragment to be searched in.

For example, suppose a user would like to search for "*chapter*" which contains the words "*Porter*" and "*book*". As mentioned above, with this search intention, keywords "*Porter*" and "*book*" are to be searched inside the XML fragments rooted at element "*chapter*". So element "*chapter*" is used to formulate the part "*/XPath-*" in our proposed XPath-based keyword search. Without the knowledge of XML structure and complete knowledge of XPath expression, user can easily combine descendant axis, "*//*", with the keyword "*chapter*". Then, user can easily combine the resulted part XPath "*//chapter*" with the keywords "*Porter*" and "*book*", which results in "*//chapter[ftcontains(Porter book)]*". Such combination can ensure that the keywords "*Porter*" and "*book*" will be searched only inside the sub-trees rooted at element "*chapter*". It is worth to mention that the above search intention can also be expressed by using XPath Full Text [30], but the syntax of XPath Full Text is very complicated and much far away from the concept of user-friendliness. For example, to address the above search intention, the user must know whether the keywords "*Porter*" and "*book*" are textual values or XML elements, and if they are textual values, the knowledge of which XML elements contain them is required. This requires complete knowledge of XML structure and syntax of XPath Full Text. Moreover, XPath Full Text is not applicable in XML streams.

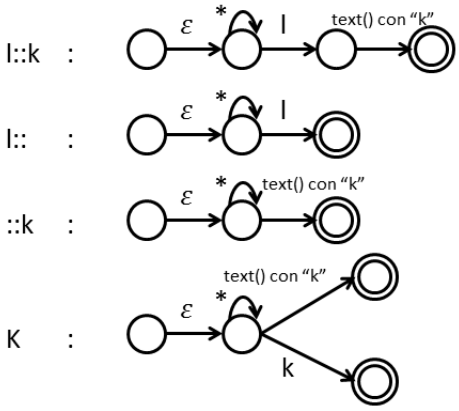


Figure 4.1: Basic Extended-NFA Location Steps.

4.1.3 Extension of NFA Model in YFilter

Our aim to process XPath-based keyword search is to find XML sub-trees that satisfy XPath-part before checking if those sub-trees satisfy keyword constraint. Therefore, the naive solution is to process all XPath- parts of all queries first by using NFA of YFilter [35]. Then, check if the found sub-trees match the keyword constraint later by using query index of CKStream [26], which contains all unique keywords of all queries. This is not efficient because when XML sub-trees are found to be matched to the XPath- parts of some queries, only the keywords of those queries should be checked. Therefore, to avoid checking unrelated keywords, both XPath- part and keyword-search part should be checked together by using NFA of YFilter. However, since NFA of YFilter only supports XPath, we extend NFA of YFilter so that it also supports XPath-based keyword search as follow.

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. Since keyword(s) in keyword search can appear either in labels of XML nodes or textual contents of nodes, and can be matched to any label or textual content in any section of XML data (similar to "//" in XPath), the extended-NFA location steps of the four types of keywords of keyword search are to be started with the "ϵ" epsilon-transition and "*" transition in this order. The edges with the labels of the form "*text() con*" represent state transitions corresponding to textual contents of nodes, which are triggered when textual contents of nodes contain the specified keywords. In addition, each accepting state in the extended NFA contains the position of bit inside the query bitmap and the matched query ID. Any query is detected as matched if all of its bits in the query bitmap are true. The basic extended-NFA location steps are shown in Figure 4.1. In this figure, nodes with double line are the accepting states.

4.1.4 Combining YFilter and CKStream

YFilter [35] is famous for efficiently processing large number of XPath expressions against the incoming XML streams while CKStream [26] is able to process multiple keyword search with acceptable throughputs. Therefore, to enable XPath-based keyword search over XML streams, it is important to combine the above two methods. By using the extended NFA model, our proposed work constructs a single extened-NFA corresponding to a set of XPath-based keyword search. Similar to YFilter and CKStream, the proposed scheme is based on SAX parser [14] to parse the incoming XML streams. In addition, two data structures, query bitmap (`query_bitmap`) and set of used queries (`used_query`), are borrowed from CKStream to process XPath-based keyword search; they are used whenever there is a push of an entry into the stack. When there is a trigger to pop an entry from the stack, queries are evaluated to be matched or not by looking at bits in `query_bitmap` corresponding to queries in the set `used_query`.

Single extended-NFA: A single extended-NFA is constructed from the set of XPath-based keyword search by using the four basic extended-NFA location steps as shown in Figure 4.1. The accepting states contain two main pieces of information, the position of bit in the query bitmap corresponding to the keywords and the query ID to which the keyword matches. Different from YFilter, when the accepting state is reached, it does not mean that the corresponding queries match, but instead it will get the bitmap position and IDs of

queries, which will be used to update the query bitmap and the set of used queries as we explain below.

Stack: The stack in our proposed method stores entries corresponding to SAX events that are already processed. Different from the entry of the stack in CKStream, this entry contains only three main pieces of information: 1) set of state IDs, 2) query bitmap, 3) set of used queries.

Query bitmap: A query bitmap contains all the bit corresponding to all unique keywords being processed. Different from CKStream, the bitmap positions of the matched keywords will be obtained from the single extended-NFA when the accepting states are reached, and those bitmap positions are used to set the corresponding bits in the query bitmap to true. Any query is evaluated to be matched or not by checking if all of its bits in the query bitmap are true. This checking is done periodically when the event `endelement(tag)` is processed.

Set of used queries: This set contains the IDs of all queries whose keywords match the incoming events of XML streams. These query IDs are obtained when accepting states are reached in the single extended-NFA when processing the incoming events of XML streams. When the system processes the `endelement(tag)`, only the queries whose IDs exist in the set of used queries will be checked. This helps reduce unecessary checking and consequently can speed up the processing time and increase throughputs of the system.

The detail of our proposed algorithm are shown in Algorithm 1 below. There are three main blocks, **callback function at the start of element**, **callback function text**, and **callback function at the end of element**.

Callback Function Start of Element: When the event `startelement(tag)` is called and processed, this function is invoked. All bits in the query bitmap are set to false by default. The `startelement(tag)` is processed against the single extended-NFA. Then all 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 is pushed into the stack.

Callback Function Text: When the event `character()` is called and before processing, 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 event `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 to the users or applications. If not all bits of the corresponding query are true, then that corresponding query ID 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. Notice that, such

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,, 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
```

appending of query bitmap and set of used queries of the popped entry to those of top entry is stopped when state connecting XPath- part to keyword-search- part is reached because the searched results should be the sub-trees of XML fragments that satisfy the XPath- part.

To fully explain our proposed method, we show how this method works by running the two queries below against the XML data shown in Figure 2.1. The detail run is shown in Figure 4.3.

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 extended-NFA is constructed as shown in Figure 4.2.

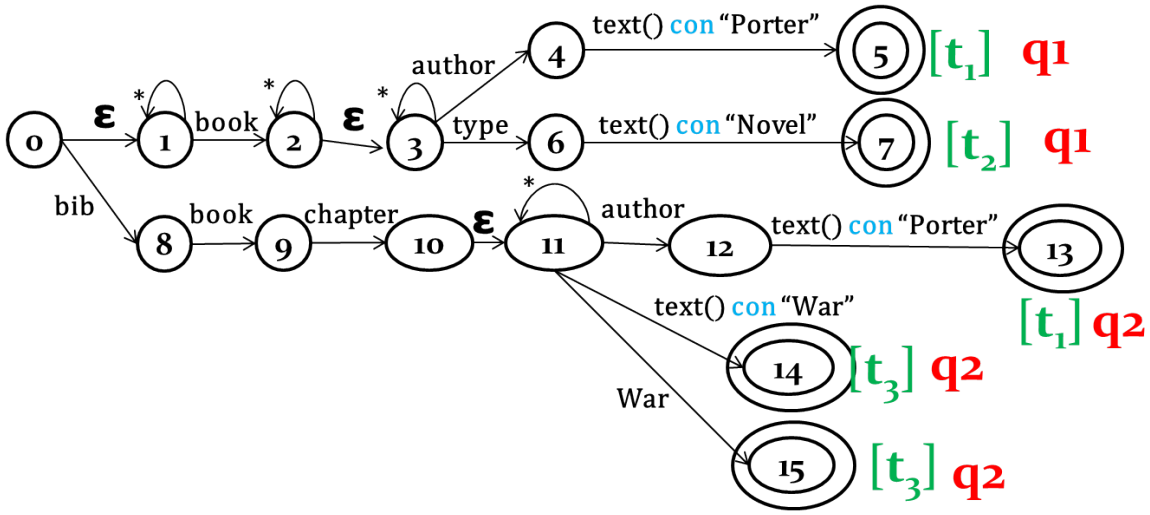


Figure 4.2: A single extended-NFA

We will show how our proposed method works by running these queries against the XML data shown in Figure 2.1. Figure 4.3 shows the detail of our running example. As explained above, when processing any XML element, three main pieces of information are generated. They are a set of states, a `query bitmap`, and a set of `used_query`. They are packed in an entry and pushed into the stack as shown in 4.3.

As shown in 4.3(a), when any 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 it into the stack. If the element is associated with textual content, on receiving a `character()` event, the system follows the same procedure to get the newly active states. In this case when element <bib> is called, NFA-look up is done against our NFA in Figure 4.2 to get the newly active states. The newly active states are states "*1, 8*". Since state "*1*" and "*8*" are not the accepting states, a blank *query bitmap* and set *used_query* are packed with the set of states "*1, 8*" into an entry and pushed into the stack. It follows the same procedure when processing element <book> and <author>. When processing element <author> together with its textual content against the NFA shown in Figure 4.2, the newly actives states are "*1, 2, 3, 4, 5*". Since state "*5*" is the accepting state, the bit position of `query bitmap` and query ID are obtained and used to set its `query bitmap` and set `used_query` respectively.

In this case, we get the bit positon "t_1" and query ID "q_1" and update the `query bitmap` and set `used_query` respectively as shown in 4.3(a).

Figure 4.3(b) shows the detail when process event `endelement(tag)`. Upon receiving an event `endelement(tag)`, the entry is popped out of the stack. Then it checks the set `used_query` of the popped entry. If the set `used_query` is not empty, *IDs* of queries in the set `used_query` are used and it checks the bitmaps of the corresponding queries in the `query_bitmap` of the popped entry. If not all bits in `query bitmap` of any queries are true, the respective queries are not matched. In Figure 4.3(b), when the end element `</author>` is called, it first checks the set `used_query` and obtains the query ID q_1. Then it checks the corresponding bits of query q_1 in the `query bitmap`. Since not all bits of this query are set, q_1 is not matched. Since no query is matched, the bits of the corresponding unmatched queries (q_1) are used to update the `query bitmap` of the ancestor entry by using `"OR"` operator. And the IDs of the unmatched queries(q_1) is inserted into the set `used_query` of the ancestor entry. In this case the `query bitmap` of the ancestor entry is set at position "t_1" and query "q_1" is inserted into the set `used_query` of the ancestor entry.

After processing `<type>`, `</type>`, `<title>`, `</title>`, `<chapter>`, `<author>`, `</author>`, `<title>`, `</title>`, the stack is shown in Figure 4.3(c) by following the same procedure as explained above.

In Figure 4.3(d), when the end of element `</chapter>` is called, it follows the same procedure as that in Figure 4.3(b). However, this time, all bits corresponding to query q_2 are set, so q_2 is matched. Therefore, all related information of q_2 is not used to update the ancestor entry. Finally, the entry of element *chapter* is popped of the stack, and as a result, the sub-tree rooted at element *chapter* is returned as a result.

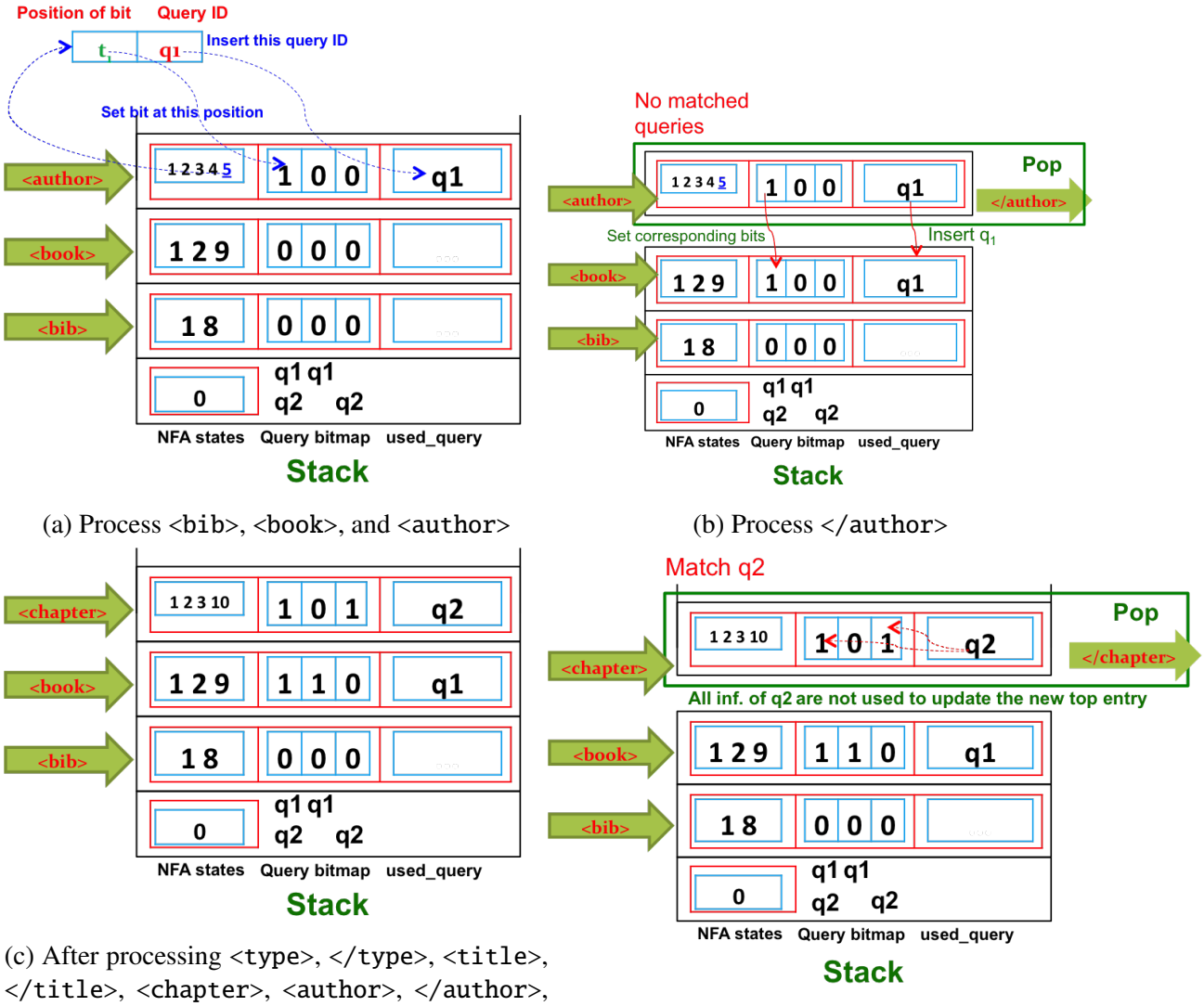
4.2 Experimental Evaluation

4.2.1 Setup

The algorithm was implemented using Java based on the existing YFilter [35] and the SAX API [14] from Xerces Java Parser. All data structures, query bitmaps and sets of used queries, were kept entirely in memory. All experiments were performed in an Intel Core 2.33GHz machine with 2 GB of memory in Windows XP Service Pack 2 except experiments in Sections 4.2.3 (on DBLP dataset) and 4.2.4, which were performed in a 2GHz Intel Core i7 machine with 8 GB of memory in MacOS Sierra. Notice that we used two new machines because the old machine used too old OS (Window XP) and was not available when doing new experiments. The experiments that were done on different machine were independent from each other, so they did not have any impact on the evaluation of the proposed approach and comparative approaches.

We used two types of datasets, synthetic data and real data. The synthetic data was generated by the xmlgen of XMark [38]. The real datasets are DBLP-biographic information on major computer science journals and proceedings [56], and Mondial-world geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources [39]. The detail of the three datasets are presented in Table 4.1.

We generated the sets of XPath-based keyword searches using data from each dataset.



(a) Process <bib>, <book>, and <author>

(b) Process </author>

(c) After processing <type>, </type>, <title>, </title>, <chapter>, <author>, </author>, <title>, and </title>

(d) Process </chapter>

Figure 4.3: A running example of the proposed method

Since there is a performance impact when searching for different kinds of keywords (*l::, k, ::k, l::k*) in our system, we separately generated the sets of queries which contain only keywords in the forms of *l::, k, l::k*, and *::k*. And we also generated the sets of queries which randomly contained keywords in the above forms. For example, we could generate: `/site/regions/australia[ftcontains(::begin ::administer ::caius ::whose)]`, whose keywords are in the form of *::k* from XMark dataset. Moreover, since the same keywords can appear several times in different queries, that can affect the performance of our algorithm. We divided our sets of queries into two categories, the sets of queries in which the same keywords can appear in different queries in the same set and the sets of queries in which the same keywords cannot appear in different queries in the same set.

We measured elapsed time for processing each dataset. This included the time spent to create the extended-NFA, the query bitmaps, and the sets of used queries. Similarly, we measured the average memory usage and the number of extended-NFA states while processing each dataset.

Table 4.1: All datasets used in the experiments

Dataset	Element	Attributes	Max-depth	Avg-depth
XMark [38]	333 millions	333 millions	5	3
DBLP [56]	3,332,130	404,276	6	2.90
Mondial [39]	22,423	47,423	5	3.59

4.2.2 Scalability

For this experiment, we investigated how well the proposed approach could handle the processing of XPath-based keyword search over XML streams. For this purpose, we varied the number of queries and query keywords, then investigated the increase in number of NFA states, memory usage, and filtering throughputs. Notice that for this experiment, MLCA [36] is adopted.

Varying the Number of Queries

First, we investigated the impact on the performance of the algorithm when the number of queries increased.

We varied the number of queries from 1, 10, 100, 200, 400, 600, 800, and 1000. 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 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, and 4.18. This is because the system needs to process each single query and output the result of each query. Figures 4.4, 4.5, and 4.6 show that, in all datasets, 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. This means that same keywords appear in different queries very frequently and the number of unique labels is very limited. We could not generate 100 or 1000 queries in which the same keyword appears only once in a particular query in the set of queries. As a result, the increase of memory usage and the decrease of throughputs when the number of queries increases do not change much.

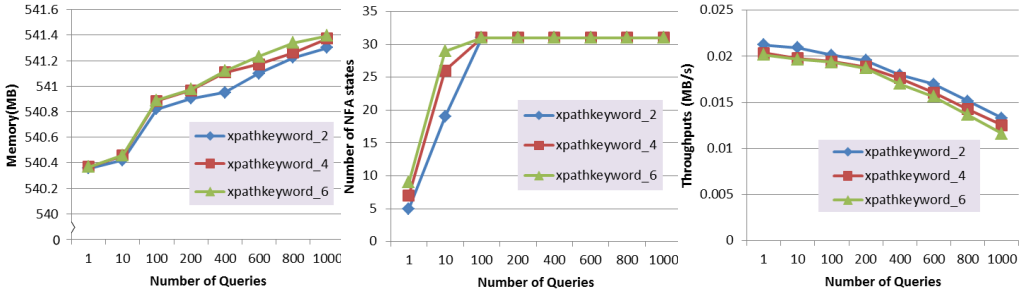
Varying the Number of Query Terms

Next, we investigated the impact on the performance of our algorithm when we increased the number of search terms from 2, 4, and 6. 6 keywords are a reasonable limit when one uses to specify the query [26]. We first randomly generated the sets of queries whose same keywords can appear in more than one query as shown in Figures 4.4, 4.5, 4.6. With these sets of queries, we observed that when the number of keywords increases, the memory usage, the number of extended-NFA states and throughputs do not change much between 2, 4, and 6 keywords. These caused by the more frequency that same keywords appear in different queries in the same set as shown in Figures 4.4, 4.5, and 4.6. As mention above, since the number of unique XML elements (label) of DBLP is very limited (at around 31), the number of NFA states for DBLP becomes constant at around 31 states even though we increase the number of queries and query terms as shown in Figure 4.4. Similarly, in Figure 4.5, the number of NFA states of Mondial dataset becomes constant at around 14 states and the

number of NFA states of XMark dataset becomes nearly constant at around 240 states shown in Figure 4.6. Because of these reasons, the memory usages and the throughputs do not change much between 2, 4, and 6 keywords when we increase the number of queries.

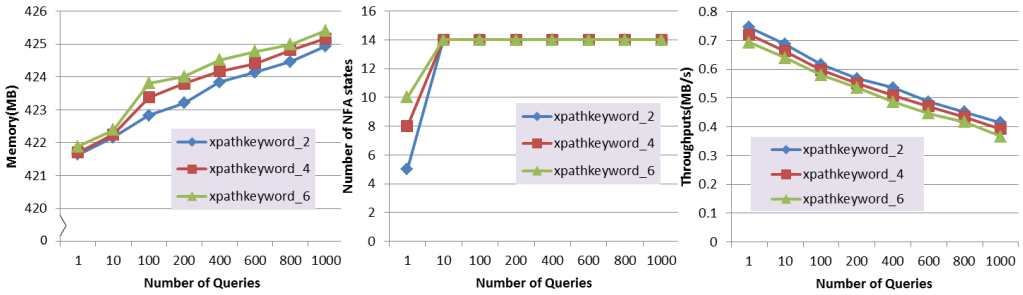
Next we generated sets of queries in which same keywords appear only in one query in the same sets. As shown in Figures 4.7, 4.8, and 4.9, we generate the XPath-based keyword searches, whose keyword is in the form "*k*", which can be matched to either the label or the textual value of XML nodes. Next, we generate the XPath-based keyword searches whose keywords are in the form l::k (a keyword contains both label and textual value of XML nodes) as shown in Figures 4.10, 4.11, and 4.12. And similary, the XPath-based keyword searches which are in the form of mixed(*k, l::, l::k, ::k*) and in the form of *::k* are generated and used as shown in Figures 4.13, 4.14, 4.15, 4.16, 4.17, and 4.18 respectively.

Then we investigated the impact of the increase of unique keywords in the queries 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 thoughputs decreases significantly as shown in Figures 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, and 4.18. Though the memory usage and throughputs of the algorithm have some degradation when the number of queries and the number of keywords increases, the algorithm scales well with such increases (the rate of degradation of throughputs is much smaller comparing to the rate of the increase in both number of queries and keywords).



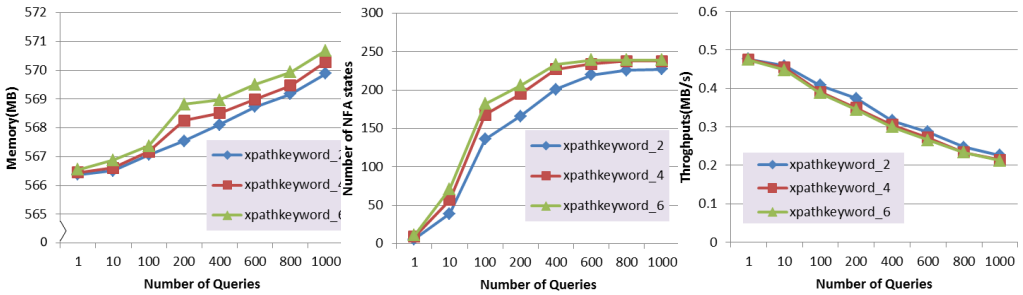
(a) Memory usage (b) Number of NFA states (c) Throughputs

Figure 4.4: DBLP: Varying the number of queries and keywords of type l::



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

Figure 4.5: Mondial: Varying the number of queries and keywords of type l::

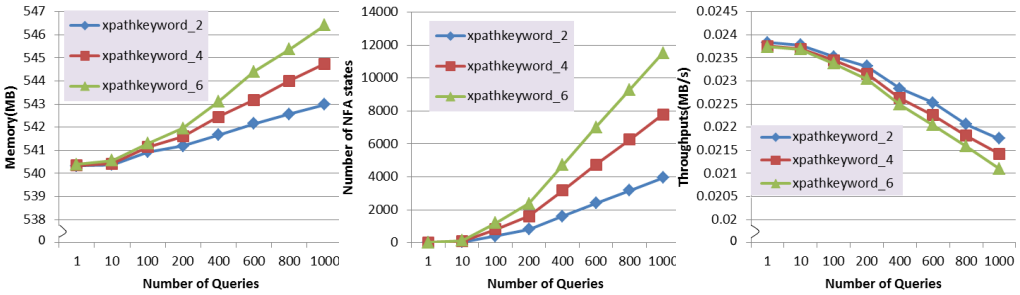


(a) Memory usage (b) Number of NFA states (c) Throughputs

Figure 4.6: XMark: Varying the number of queries and keywords of type l::

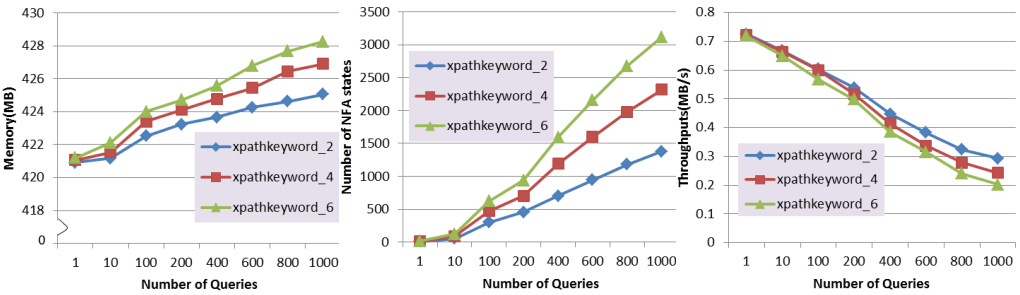
4.2.3 Accuracy

Next, we comapared the effectiveness of our proposed method with CKStream when querying for the same search intentions. For experimental purpose, we created some search intentions, then translated them into pure keyword search and XPath-based keyword search. Pure keyword search was processed by CKStream, and XPath-based keyword search was processed by the proposed approach. Since the work in CKStream [26] was implemented using SLCA [34] heuristic, we separately implemented the proposed method with SLCA [34] heuristic.



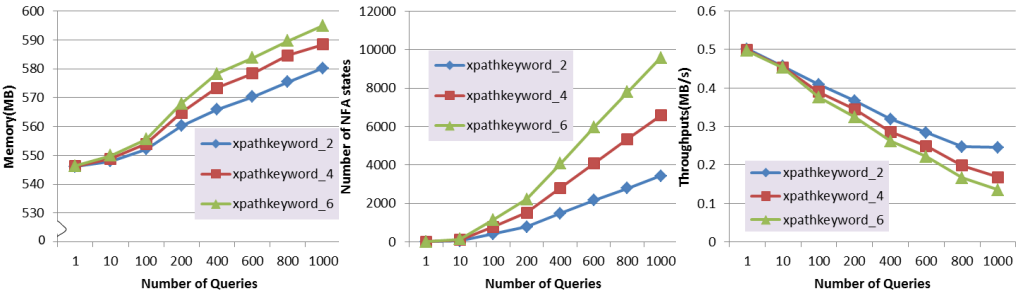
(a) Memory usage (b) Number of NFA states (c) Throughputs

Figure 4.7: DBLP: Varying the number of queries and unique keywords of type k



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

Figure 4.8: Mondial: Varying the number of queries and unique keywords of type k



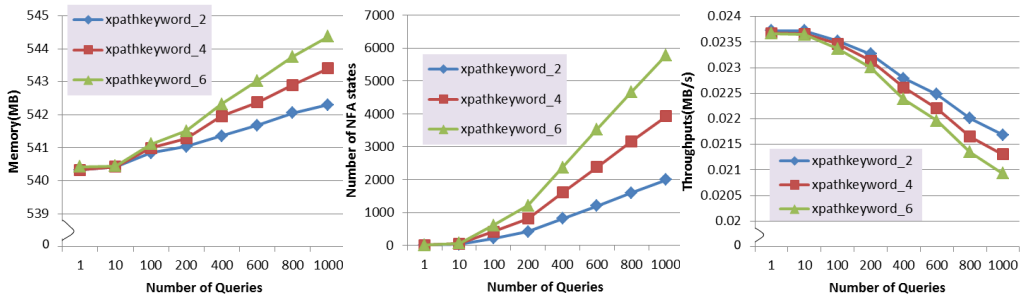
(a) Memory usage (b) Number of NFA states (c) Throughputs

Figure 4.9: XMark: Varying the number of queries and unique keywords of type k

The queries used for evaluation are listed in Tables 4.2 and 4.3 for XMark, and Tables 4.4 and 4.5 for DBLP.

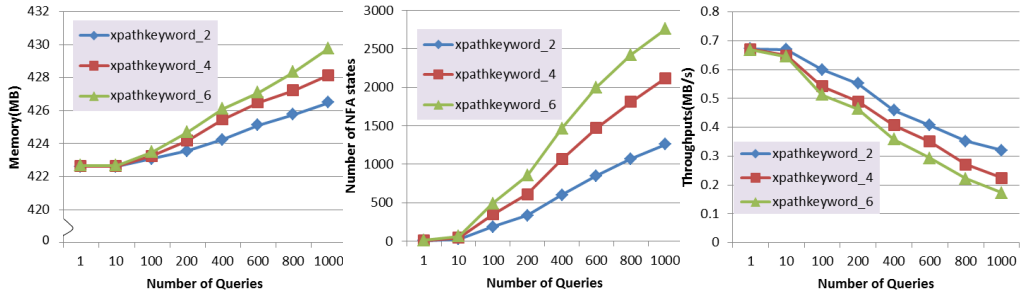
The relevant matches of the search intentions shown in Table 4.2 were chosen to be the sub-trees rooted at elements "*auction*", "*shipping*", "*category*", "*people*", and "*regions*", which contained the respective keywords. Notice that, these sub-trees are big sub-trees that cover large part of XMark dataset. They were chosen because it was difficult to make meaningful search intentions by using small sub-trees in XMark.

The relevant matches of the search intentions shown in Table 4.4 were chosen to be the sub-trees rooted at elements "*chapter*", "*journals*", "*conference*", "*masterthesis*", and "*abstract*", which contained the respective keywords. These sub-trees were small sub-trees about new publications.



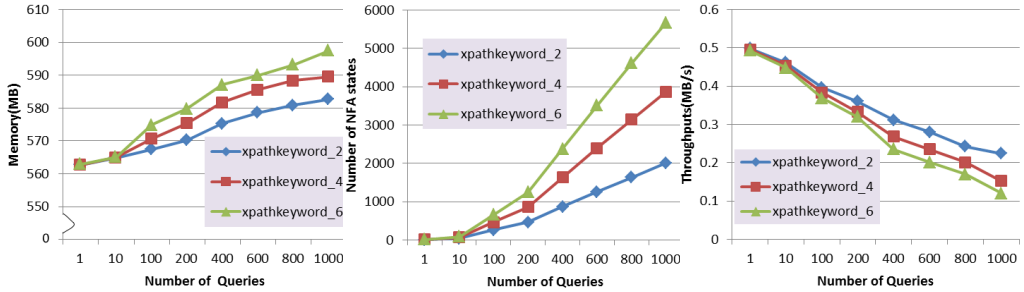
(a) Memory usage (b) Number of NFA states (c) Throughputs

Figure 4.10: DBLP: Varying the number of queries and unique keywords of type l::k



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

Figure 4.11: Mondial: Varying the number of queries and unique keywords of type l::k



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

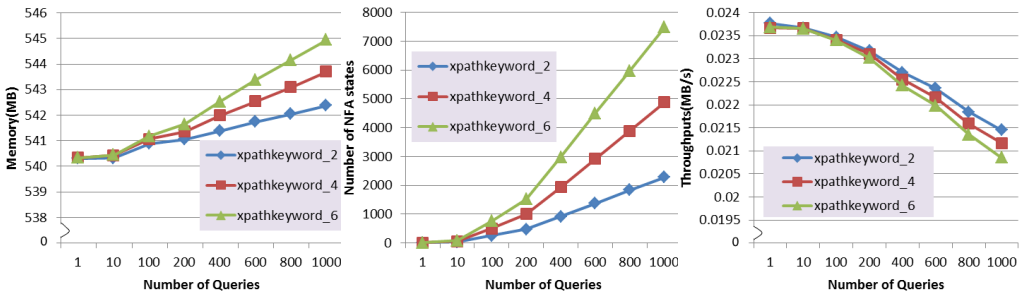
Figure 4.12: XMark: Varying the number of queries and unique keywords of type l::k

We investigated how the size of search boundaries, which are defined by the above subtrees, affected the performance of the proposed approach.

For the experimental purpose, we modified XMark dataset and DBLP dataset by adding several keywords into various textual elements randomly. Moreover, for DBLP dataset, subtrees rooted at elements "*chapter*", "*journals*", "*conference*", "*masterthesis*", and "*abstract*" were added to control the total number of wanted results.

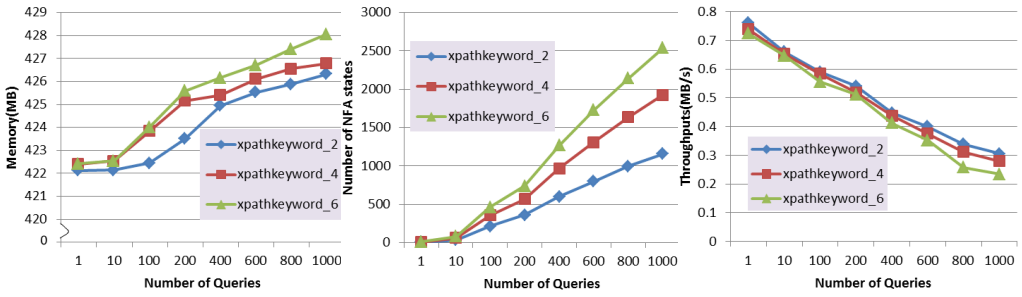
F-Measure

Next, we evaluated the effectiveness of our proposed method and CKStream based on precision, recall, and F-measure. Precision is the percentage of retrieved results that are desired



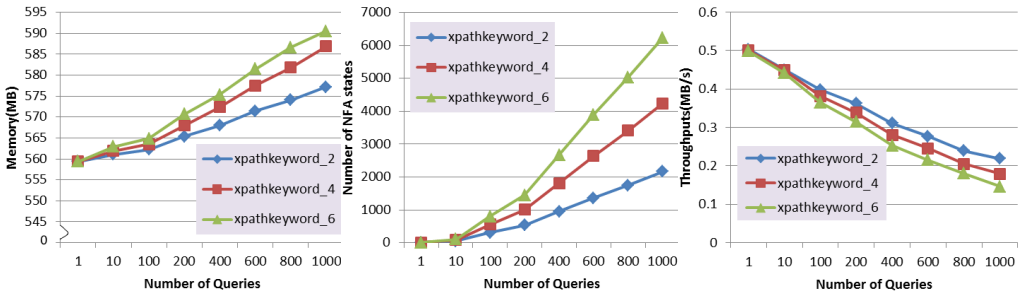
(a) Memory usage (b) Number of NFA states (c) Throughputs

Figure 4.13: DBLP: Varying the number of queries and unique keywords of type l::, ::k, k, l::k



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

Figure 4.14: Mondial: Varying the number of queries and unique keywords of type l::, ::k, k, l::k

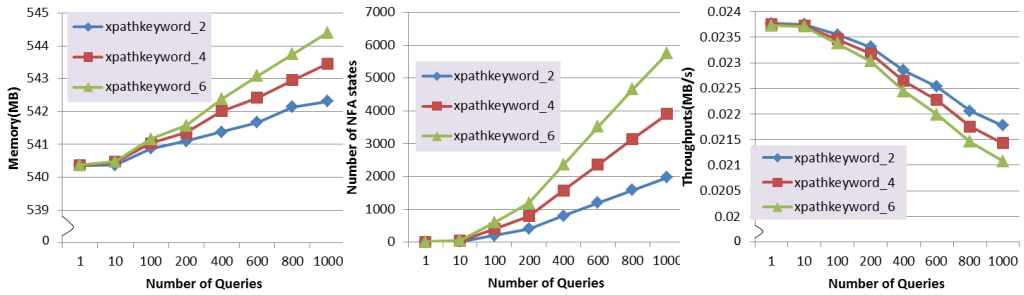


(a) Memory usage (b) Number of NFA states (c) Throughputs

Figure 4.15: XMark: Varying the number of queries and unique keywords of type l::, ::k, k, l::k

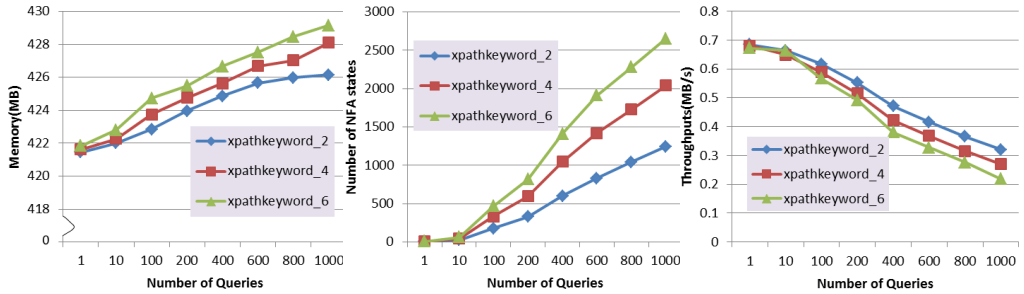
by users. Recall is the percentage of relevant results that can be retrieved. F-measure is the weighted harmonic mean of precision and recall.

As shown in Figures 4.19(a) and 4.20(a), CKStream has very low precision on all queries for both datasets because it returns many unrelated results (any sub-trees that contain all keywords). Whereas, our proposed method has high precision (100%) because it only returns the relevant matches, which are the sub-trees rooted at elements "*auction*", "*shipping*", "*category*", "*people*", and "*regions*" for XMark, and the sub-trees rooted at elements "*chapter*", "*journals*", "*conference*", "*masterthesis*", and "*abstract*" for DBLP, which contain the re-



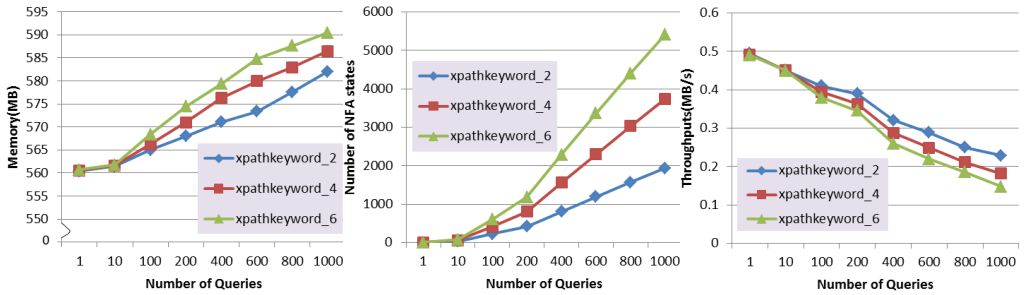
(a) Memory usage (b) Number of NFA states (c) Throughputs

Figure 4.16: DBLP: Varying the number of queries and unique keywords of type ::k



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

Figure 4.17: Mondial: Varying the number of queries and unique keywords of type ::k



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

Figure 4.18: XMark: Varying the number of queries and unique keywords of type ::k

spective keywords. This proves the effectiveness of the application of the part "`/XPath-`" of our proposed method, which can greatly help reduce the vagueness of keyword search. Both methods have high recall (100%) as shown in Figures 4.19(b) and 4.20(b) because they both return all desired results for both datasets.

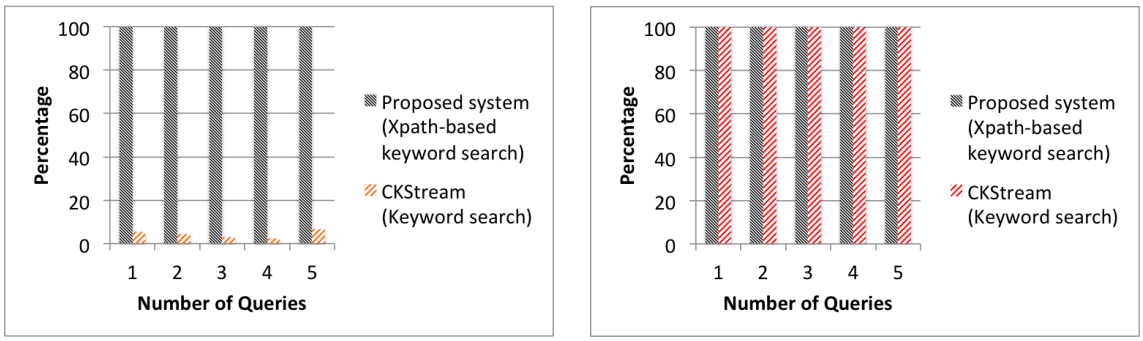
We calculated the average F-measure as shown in Table 4.6. It can be seen that our proposed method achieves much higher F-measure than CKStream because CKStream returns many undesired results as explained earlier.

Table 4.2: XMark: Search intentions and all translated keyword searches

N.	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*

Table 4.3: XMark: The translated XPath-based keyword searches from search intentions shown in Table 4.2

N.	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)]



(a) Precision (b) Recall

Figure 4.19: XMark: Precision and Recall

Performance Comparision

Next, we evaluated the performance of our proposed method and CKStream based on their throughputs and memory usage.

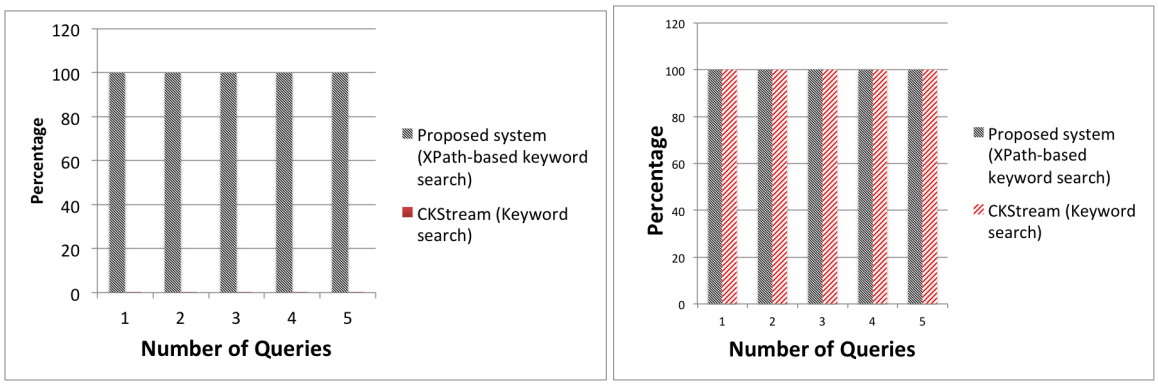
As shown in Figures 4.21 and 4.22, our proposed method produces higher throughputs and consumes less memory than CKStream for both datasets because, in our proposed method, the respective keywords shown in Table 4.3 are searched only inside the sub-trees rooted at elements "*auction*", "*shipping*", "*category*", "*people*", and "*regions*", and the re-

Table 4.4: DBLP: Search intentions and all translated keyword searches

N.	Search intentions	Keyword search
1	Find the "*chapter*" that is related with "*XML*" and "*stream*"	*chapter XML stream*
2	Find the "*journals*" that is related with "*spatial*" and "*keyword*"	*journals spatial keyword*
3	Find the "*conference*" that is related with "*graph*" and "*search*"	*conference graph search*
4	Find the "*masterthesis*" that is related with "*structure*" and "*match*"	*masterthesis structure match*
5	Find the "*abstract*" that is related with "*classification*" and "*algorithm*"	*abstract classification algorithm*

Table 4.5: DBLP: The translated XPath-based keyword searches from search intentions shown in Table 4.4

N.	XPath-based keyword search
1	//chapter[ftcontains(XML stream)]
2	//journals[ftcontains(spatial keyword)]
3	//conference[ftcontains(graph search)]
4	//masterthesis[ftcontains(structure match)]
5	//abstract[ftcontains(classification algorithm)]



(a) Precision (b) Recall

Figure 4.20: DBLP: Precision and Recall

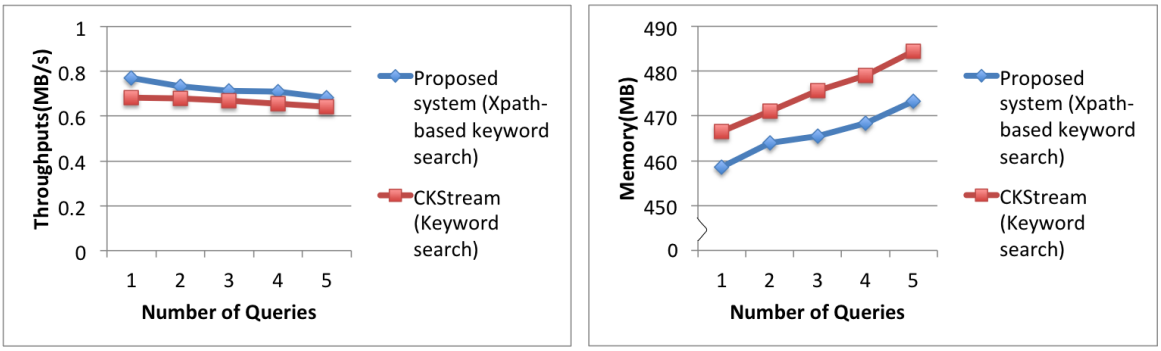
spective keywords shown in Table 4.5 are searched only inside the sub-trees rooted at elements "*chapter*", "*journals*", "*conference*", "*masterthesis*", and "*abstract*". Whereas, CKStream tries to search for all keywords shown in Tables 4.2 and 4.4 in the entire XML data of XMark and DBLP respectively. Such unnecessary searching causes the querying performance worse. As a result, our proposed method enjoys producing higher throughputs and using less memory consumption.

Notice that little improvement in throughputs and memory usage was achieved by the

Table 4.6: Comparison on F-Measure

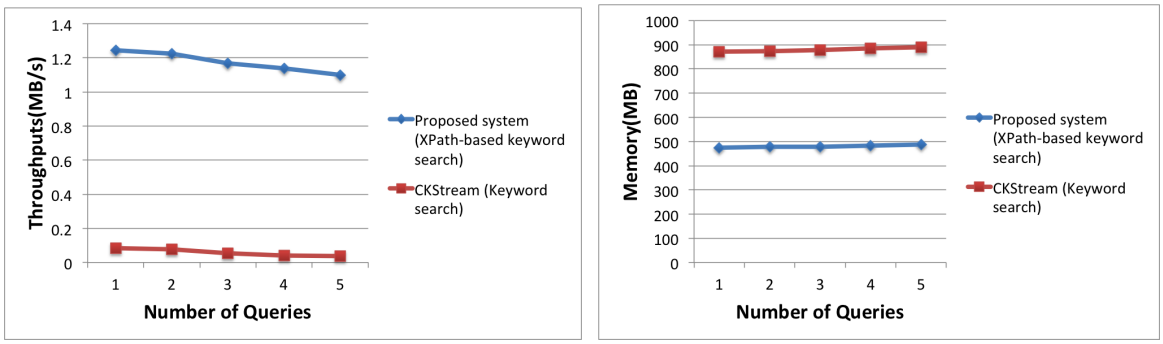
F-Measure	CKStream	Proposed Work
XMark	0.126	1
DBLP	0.1	1

proposed approach for XMark dataset because big search boundaries were defined for this dataset. However, in DBLP, the search boundaries were set to be small, so the proposed approach could achieve much improvement in both throughputs and memory usage. This results prove that the defined search boundaries can have an impact on the performance improvement of the proposed approach. Therefore, if such search boundaries are properly defined, much performance improvement can be achieved by the proposed approach comparing to CKStream.



(a) Throughputs (b) Memory Usages

Figure 4.21: XMark: Proposed system vs CKStream on queries with same search intention



(a) Throughputs (b) Memory Usages

Figure 4.22: DBLP: Proposed system vs CKStream on queries with same search intention

4.2.4 Performance Comparision on Pure Keyword Search and XPath

Next, we investigated the overheads that our proposed work could deal with pure XPath query and pure keyword search. For this experiment, we generated pure XPath and pure keyword search from the synthetic dataset XMark [38]. For XPath, the minimum depth

is 3 and maximum depth is 5. There was no "*" and "//" axis in the XPath. For keyword search, we used the set of queries with 4 keywords, and the type of keyword is in the form of "*::k*". Next, we replaced 10%, 20%, and 30% of pure XPath and keyword search by comparable XPath-based keyword search (XPath-based keyword search that gives same results as pure keyword search and XPath does). We then calculated the average throughputs of each approach. Notice that XPath100%_XPathKws0%, XPath90%_XPathKws10%, XPath80%_XPathKws20%, and XPath70%_XPathKws30% are used to refer to the set of 100% of XPath, 90% of XPath plus 10% of XPath-based keyword search, 80% of XPath plus 20% of XPath-based keyword search, and 70% of XPath plus 30% of XPath-based keyword search that are processed by the proposed approach. Similarly, Keyword100%_XPathKws0%, Keyword90%_XPathKws10%, Keyword80%_XPathKws20%, and Keyword70%_XPathKws30% are used to refer to the set of 100% of keyword search, 90% of keyword search plus 10% of XPath-based keyword search, 80% of keyword search plus 20% of XPath-based keyword search, and 70% of keyword search plus 30% of XPath-based keyword search that are processed by the proposed approach.

First, we compared the performance of CKStream [26] with our proposed work by investigating on their throughputs. As shown in Figure 4.23, the performance of the proposed approach is comparable to that of CKStream when processing a set of 100% pure keyword search. Actually, for this experiment, the proposed approach achieves a little bit less throughputs than CKStream, but the difference is very small. For this reason, their throughputs are closely overlapping in Figure 4.23, which proves that using extended NFA for keyword searching of the proposed approach is as efficient as CKStream. When replacing 10%, 20%, and 30% of pure keyword search by XPath-based keyword search, the throughputs of the proposed approach is getting better than processing 100% of pure keyword search because keyword searching of the replaced XPath-based keyword search is only done inside the XML fragments defined by the XPath- part rather than in the entire XML streams.

Next, we compared the throughputs of YFilter [35] with our proposed work as shown in Figure 4.24. As can be seen, when processing the set of 100% pure XPath, the proposed approach achieves comparable throughputs comparing to YFilter. This is because, when processing pure XPath, the extended NFA of the proposed approach is exactly the same as NFA of YFilter, which neither query bitmap nor set of used queries are used. When replacing 10%, 20%, and 30% of pure XPath by XPath-based keyword search, the throughputs of the proposed approach is getting worse than processing 100% of pure XPath because in addition to finding the XML sub-trees matching the XPath- part, the proposed approach needs to check if those XML sub-trees satisfy keyword search- parts of the replacing XPath-based keyword search.

The above experimental results prove that the proposed approach can achieve comparable throughputs as CKStream and YFilter when processing pure keyword search and XPath, and it is also more flexible and able to deal with more varieties of query types. YFilter can only process XPath, and CKStream can only handle keyword search. But 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.

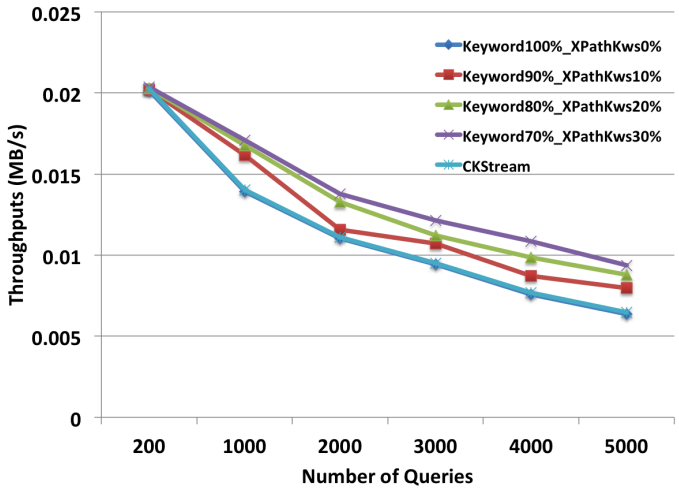


Figure 4.23: Throughputs of CKStream vs our proposed work

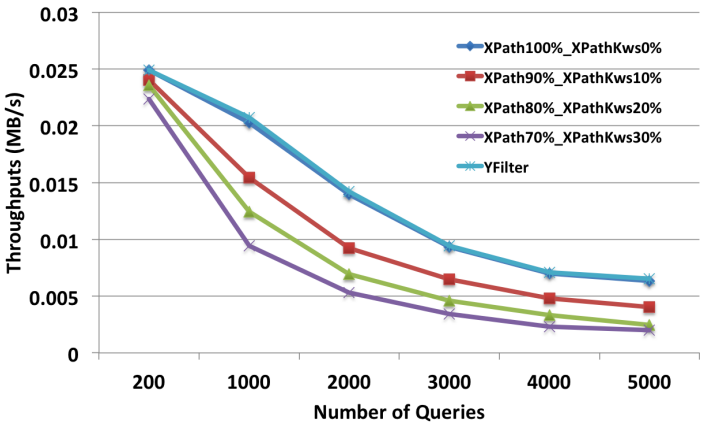


Figure 4.24: Throughputs of YFilter vs our proposed work

4.3 Summary of this Chapter

In this chapter, we have proposed XPath-based keyword search that allows users to specify their search intentions more effectively. We have also proposed a mechanism to process this query over XML streams. Different from the existing algorithms that only support one specific type of query, our proposed method supports XPath, keyword search, and XPath-based keyword search. For this purpose, 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 the proposed method 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.

Chapter 5

Keyword Search over Relational Streams

5.1 Proposed Approach

5.1.1 Overview

Our proposed framework for keyword search over relational streams has the same general process as the existing approaches [59, 61] that involve in two main steps as explained in the Section 3. The first step is to create all CNs from the given keyword search and schema of relational streams. Then, a query plan is created by combining all CNs together for efficient processing. In the second step, the query plan is directly evaluated against the incoming relational streams to find the query results. Our proposal is to create a better query plan. For this purpose, we propose a novel query plan representation, called *MX-structure* (maximal-sharing structure), that combines all CNs by consolidating all common edges. By using MX-structure, we can avoid redundant nodes and edges to be expanded. To enable the processing of MX-structure over relational streams, we introduce fine-grained node buffers and branch maps for managing existing partial/full query results. To deal with expiration of tuples, we adopt *lazy approach* [59] where expired tuples are removed when node buffers are probed.

5.1.2 MX-Structure

First, we introduce the proposed MX-structure by starting from its construction as follow. First, each CN is marked by one unique ID, which is used to detect its matched MTJNTs. In each CN, the root node (and the output node as well) is determined as the centered node (the node with minimal path to all leaf nodes). Then, all CNs are merged in such a way that all edges are unique; e.g., edges in MX-structure are created only for different combinations of nodes regardless of their positions (root or leaf). Such information needs to be maintained as well. In the sequel discussion, we denote by `()` a leaf node and by `[]` a root node. Notice that, in MX-structure, each source node and each edge represent selection operation and join operation between two connected nodes, respectively.

The pseudo code to construct MX-structure is shown in Algorithm 2. Basically, all CNs are added to an MX-structure one by one. When adding a new CN, we take each edge and check its existence; we add one only if it has not been added yet. Next, the ID of CN is added to each of its edges in MX-structure. The information about each CN's root and leaf nodes

Algorithm 2 MX-structure Construction

Input: *CNs*

```
1: Initialize MX-structure MX
2: while each CN do
3:     while each edge do
4:         if edge not exists in MX then
5:             add that edge into MX
6:         end if
7:         add id of that CN of that edge in MX;
8:         if each node is either root or leaf node then
9:             add id of that CN into MX to mark leaf or root node of CN.
10:        end if
11:    end while
12: end while
```

is also maintained.

Figure 5.1 illustrates an example of MX-structure for all CNs in Figure 2.14 (Notice that all CNs in Figure 2.14 are generated for keyword search "k_1, k_2" on relational streams whose schema is shown in Figure 2.11). Nodes marked with double lines show root nodes, and black nodes are leaf nodes. The label on each edge represents the set of corresponding CNs in term of IDs. The numbers in `()` and `[]` are the IDs of CNs of leaf and root nodes, respectively.

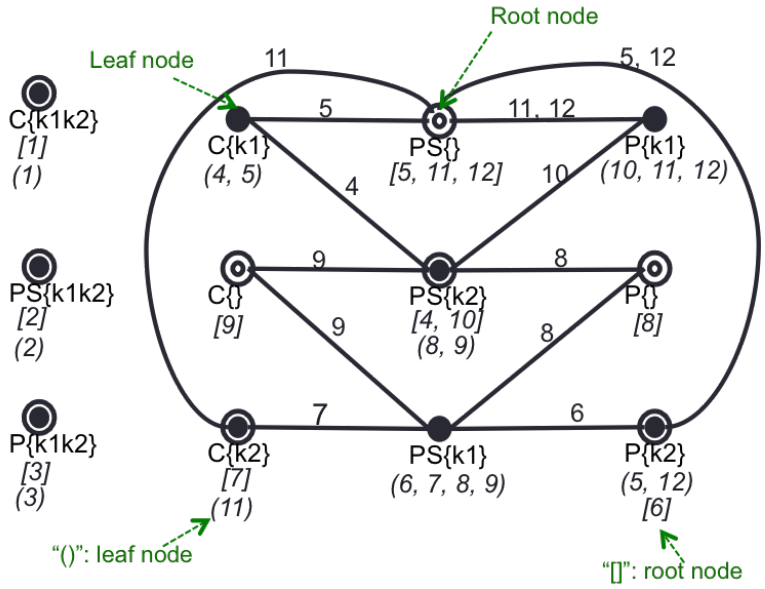


Figure 5.1: MX-structure for all CNs in Figure 2.14

5.1.3 Query Evaluation in MX-Structure

To evaluate queries over relational streams using MX-structure, we need to track the matching status of each tuple to the respective CNs. For example, let us look at Figure 5.1. If all joins between all edges (*P{k1}*-*PS{}*-*P{k2}*) of CN 12 are detected, tuples that contribute to form MTJNT of CN 12 need to be output as a query result. This is allowed by the fine-grained status management of (existing) tuples using *node buffers*. More precisely, for each

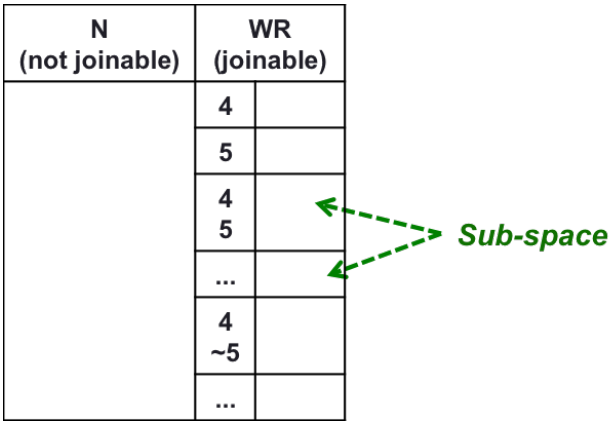


Figure 5.2: Node buffer of node $C\{k1\}$ of MX-structure in Figure 5.1

incoming tuple, its join-ability is checked according to the *probing sequence*, and is stored in an appropriate sub-space in a sub-buffer w.r.t. the corresponding CN, which is allocated dynamically when necessary. Thus the proposed scheme achieves better performance while consume less memory space.

Node Buffers

In an MX-structure, each node buffer is divided into two sub-buffers, N and WR. Sub-buffer N is for storing tuples that are *not joinable*, while WR is for storing tuples that are *joinable* with other tuples. Moreover, sub-buffer WR is divided into sub-spaces according to the CNs it belongs to. Each sub-space indicates the joint status of each joinable tuple to its matched CNs. In the following discussion, we denote $\sim n$ as the sub-space for tuples that are *fully matched* (as part of the complete query results) w.r.t. CN n, whereas n as the sub-space for tuples that are *partially matched* (not part of the complete query results) w.r.t. CN n. The table in Figure 5.2 shows the buffer of node $C\{k1\}$ of MX-structure in Figure 5.1. As can be seen, node $C\{k1\}$ appears in CNs 4 and 5. For this reason, some sub-spaces are created in sub-buffer WR; e.g., {4, ~5} is for those tuples that partially match in CN 4 and fully match in CN 5. Notice that we dynamically create sub-spaces when necessary to avoid the allocation of unnecessary (unpopulated) sub-spaces.

Probing Sequence

To systematically evaluate queries, for each incoming tuple, we check its joinability with other existing tuples in the node buffers in other child and/or parent nodes, and such probes are performed in the leaf-to-root direction; if a new tuple arrives at a leaf node, then we probe its parent nodes; otherwise, we first probe the child nodes, then probe the parent nodes. More precisely, when probing child nodes, we probe existing tuples in both sub-buffers N and WR if the nodes being probed are at the leaf level, but do so only in WR if the nodes are at non-leaf levels. If it turns out that the incoming tuple is not joinable with any other tuples in the node buffers in the child nodes, then current probing is finished, and the tuple is stored in sub-buffer N (not joinable); otherwise, it is stored in a sub-space in sub-buffer WR

that corresponds to the CN(s) to which the incoming tuple contributes to form the resulting MTJNT(s).

Note here that we call the CN(s) that the incoming tuple contributes to form MTJNT(s) *active CN(s)*. The set of active CNs are defined as follows:

$$cn_{active} = cn_{edge} \cap (cn_{leaf} \cup cn_{ecsubspace}) \tag{5.1}$$

where cn_{edge} is the set of IDs of CNs assigned to the connected edge(s) being traversed, cn_{leaf} is the set of IDs of CNs assigned to the leaf node(s) if the probed child node(s) is a leaf node, and $cn_{ecsubspace}$ is the set of IDs of CNs of non-empty sub-spaces in the child node. Notice that, if the probed child node is a non-leaf node, cn_{leaf} is empty. Similarly, in sub-buffer N, $cn_{ecsubspace}$ is also empty. Determining active CNs is beneficial to avoid unnecessary probings due to the fact that inactive CNs in child nodes can never be active in parent nodes. Thus, once active CNs are determined by probing child nodes, only the parent nodes that are connected via edges of active CNs are probed, thereby avoiding unnecessary probings in the upper levels.

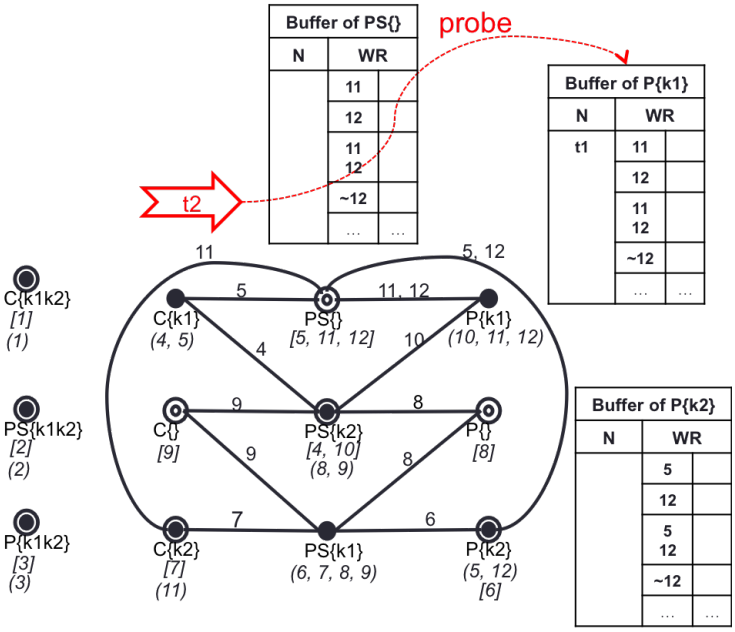
Let us look at Figure 5.3(a) as an example. Notice that only node buffers that store tuples are shown for simplicity. Let us assume that tuples *t1* and *t2*, which are 1) of tables *P* and *PS*, resp., 2) *t1* contains keywords *k1* and *t2* does not contain any query keyword, and 3) joinable with each other and arrive in this order. When *t1* arrives, we immediately probe the parent nodes *PS*{} and *PS*{*k2*}, because *P*{*k1*} is at the leaf level. As a result, it turns out that *t1* is not joinable because of empty node buffers in *PS*{*k2*} and *PS*{}, and is stored in the sub-buffer N in *P*{*k1*}. Afterwards, when *t2* arrives, we probe the child nodes, *C*{*k1*}, *P*{*k1*}, *C*{*k2*}, and *P*{*k2*}. Since buffers of nodes *C*{*k1*}, *C*{*k2*}, and *P*{*k2*} are empty, probing is ended. When probing *P*{*k1*}, *t2* turns out to be joinable with *t1* w.r.t. CNs 11 and 12.

By applying the formula explained above[1], we get $cn_{active} = \{11, 12\}$, so 1) *t1* is moved to the sub-space {*11, 12*} in sub-buffer WR, and 2) *t2* is stored in sub-space {*11, 12*} in sub-buffer WR in the respective nodes as shown in Figure 5.3(b). For subsequent probings of parent nodes, only active CNs (CNs 11 and 12) are taken into consideration. In this case, *PS*{} has no parent nodes, so probing is finished.

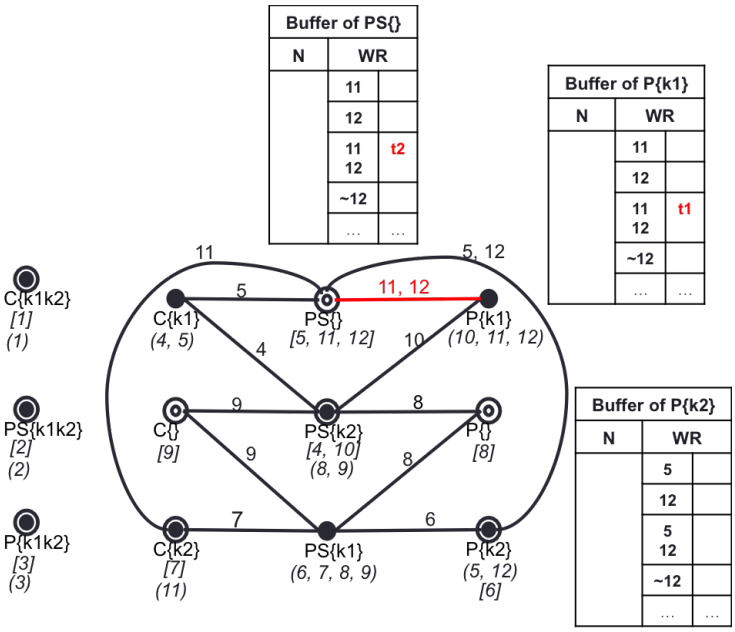
Branch Map

In the MX-structure, in many cases, root/output nodes are internal (non-leaf) nodes in one or more CNs. In addition, since probing proceeds in the leaf-to-root direction, we need to maintain for each tuple in the root node its matching status so that we can output new MTJNTs as soon as they are detected. To this end, we use a map called *branch map* to track whether there are matched tuples in all nodes from all leaf nodes up to the root/output node of any CN. More precisely, a branch map is attached to each joinable tuple in the root/output nodes. A branch map has several bits corresponding to the branches from the leaf (or leaves). When all bits are set to one, the MTJNT that contains the root tuple is output as a result. For example, node *PS*{} of MX-structure in Figure 5.1 is the root node of CNs 5, 11, and 12,

[1]We have $cn_{edge} = \{11, 12\}$ (edge *PS*{}-*P*{*k1*} belongs to CNs 11 and 12), $cn_{leaf} = \{10, 11, 12\}$ (node *P*{*k1*} is a leaf node of CNs 10, 11, and 12), and $cn_{ecsubspace} = \{\}$ (*t1* is currently in sub-buffer N). As a result, we get $cn_{active} = \{11, 12\}$.



(a) When t2 arrives, it probes $P\{k1\}$.



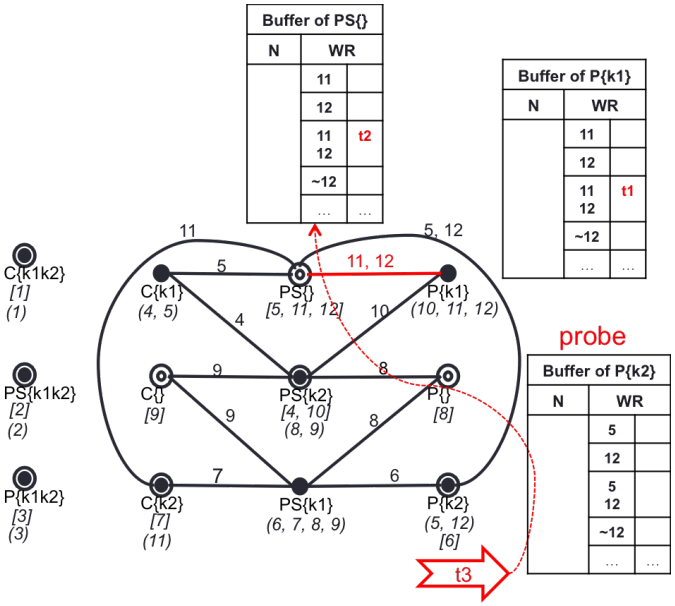
(b) *t2* can be joint with *t1*, so move them to subspace {11, 12} of their respective nodes

Figure 5.3: Example of probing sequence.

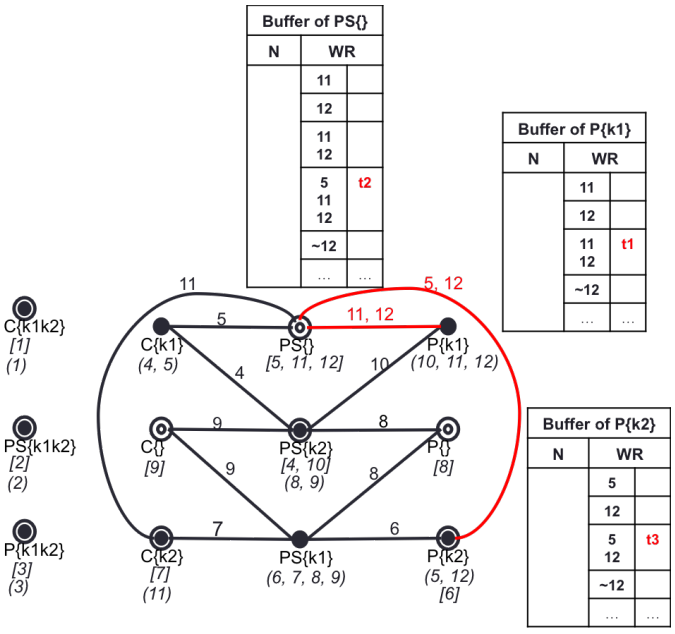
and it has two branches for each CN. Figure 5.6 shows the branch maps of this node ($PS\{\}$). Since each CN has two leaf nodes, each map has two bits which are initialized by zero.

Continued from the example in Figure 5.3(b). Since tuples t1 and t2 of edge $PS\{\}$-$P\{k1\}$ that belongs to CNs 11 and 12 are joinable, the first bits of CNs 11 and 12 corresponding to edge $PS\{\}$-$P\{k1\}$ are set to one.

Suppose tuple *t3* in $P\{k2\}$ has arrived (Figure 5.4(a)), and is joinable with *t2* w.r.t. active



(a) When *t3* arrives, it probe *PS*{}



(b) *t3* can be joint with *t2*, so move *t3* and *t2* to subspace $\{5, 12\}$ and $\{5, 11, 12\}$ of their respective nodes

Figure 5.4: Example of probing sequence

CNs 5 and 12[2]. Then, *t3* is kept in subspace $\{5, 12\}$ of node $P\{k2\}$ as explained earlier (Figure 5.4(b)). Moreover, *t2* is now joinable to CNs 5, 11, and 12, so it is moved to subspace $\{5, 11, 12\}$ of node $PS\{\}$. Since node $PS\{\}$ is the root node, the second bits corresponding to edge $P\{k2\}$-$PS\{\}$ in the existing branch map for active CNs (CNs 5 and 12) are set to one. Since all bits of CN 12 are set to one, CN 12 is detected as matched, and its matched MTJNT is returned as a query result. Then, all matched tuples are moved to the appropriate sub-

[2]We have $cn_{edge} = \{5, 12\}$ (edge $P\{k2\}$-$PS\{\}$ belongs to CNs 5 and 12), $cn_{leaf} = \{5, 12\}$ (node $P\{k2\}$ is a leaf node of CNs 5 and 12), and $cn_{ecsubspace}$ is empty (*t3* has just arrived). As a result, we get $cn_{active} = \{5, 12\}$.

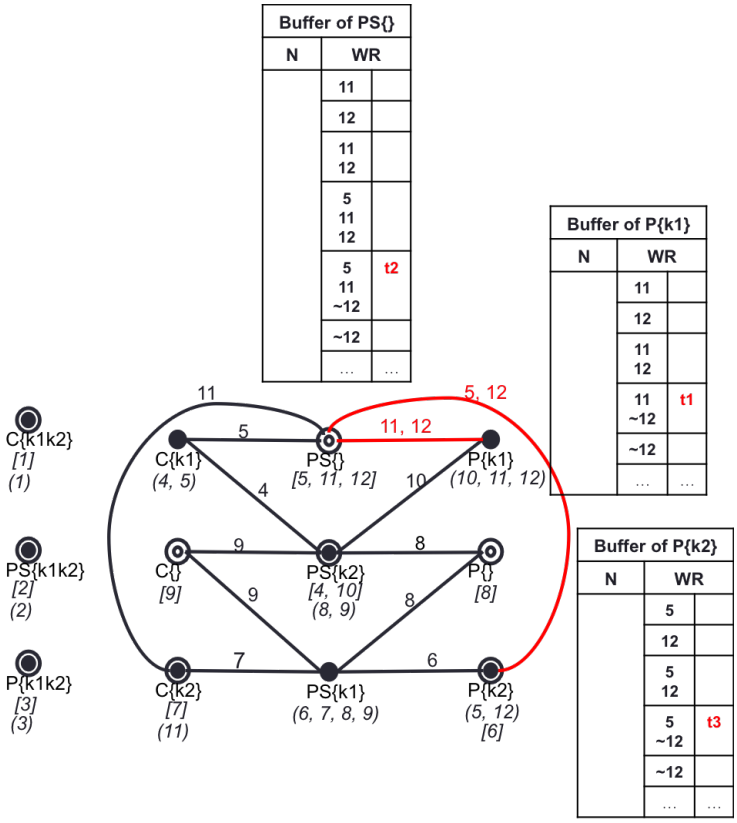


Figure 5.5: CN 12 is matched, so its MTJTNs is returned as query result. All related matched tuples are moved to the appropriate subspace.

spaces of their fully matched CN 12 as shown in Figure 5.5 for subsequent processing.

ID of CN	map	
5	0	0
11	0	0
12	0	0

Figure 5.6: Branch maps for node *PS*{} of MX-structure in Figure 5.1

Dynamic Generation of Sub-spaces

As explained earlier, we dynamically populate sub-spaces when necessary because 1) generating all possible sub-spaces requires huge memory spaces, and 2) only a few sub-spaces are used in query processing. To this end, we populate a new sub-space according to the following formula:

$$cn_{newsubspace} = cn_{oldsubspace} \cup cn_{active} \tag{5.2}$$

where $cn_{newsubspace}$ and $cn_{oldsubspace}$ are respectively the new sub-space and the existing sub-space marked by IDs of CNs for each joinable tuple. Notice that, if tuple just arrives or is currently in sub-buffer N, its $cn_{oldsubspace}$ is empty.

5.1.4 Algorithm Details

The proposed algorithm is shown in Algorithm 3. This algorithm works as follows. If the incoming tuple, t_0, belongs to a non-leaf node, it probes child nodes by calling function `Probe_child_nodes` (Line 3). This function returns *joinable_to_child* = true if there are joinable tuples in child nodes with the incoming tuple. Otherwise, it returns *joinable_to_child* = false, which results in finishing the current probing, and t_0 is stored in sub-buffer N (Line 4).

This function `Probe_child_nodes` works as follow. For each sub-space of sub-buffer WR in each child node (and sub-buffer N if child node is leaf node), cn_{active} is computed by Equation (5.1). If cn_{active} is not empty, it checks each tuples in that sub-space (Line 2–5). If there are tuples joinable with the incoming tuple, *joinable_to_child* is set to true (Line 6), and function *Match_CN* is called to check if any partially matched CNs in cn_{active} are fully matched (Line 7). This function returns *joinable_to_child* (Line 12).

In function *Match_CN*, each CN in cn_{active} is checked if there are fully matched CNs. First, appropriate sub-space, $cn_{newsubspace}$, is computed by Equation (5.2) (Line 1). Then, for each partially matched CN, *branch_map* is updated (Line 2). There are fully matched CNs if the parent node is root node and all bits in the *branch_map* are set (Line 3–4). If any fully matched CN is found, its MTJNT is returned as a result, and sub-space, $cn_{newsubspace}$, are updated according to the fully matched CN (Line 5–6). Finally, all matched tuples are moved to the appropriate sub-space $cn_{newsubspace}$ (Line 8).

Back to the main algorithm, if the incoming tuple is from leaf nodes or *joinable_to_child* is true, subsequent parent nodes are probed until no parent nodes have joinable tuples (Line 8–18) by calling function `Probe_parent_nodes` (Line 10) following similar procedure above.

Algorithm 3 MX-structure Evaluation

Input: Tuple t_0 just from streams, MX-structure MX

```
1:  joinable_to_child = false
2:  if t0 from non-leaf nodes then
3:      joinable_to_child = Probe_child_nodes (t0, MX)
4:      Put t0 in sub-buffer N if joinable_to_child = false
5:  end if
6:  if t0 from leaf nodes or joinable_to_child = true then
7:      put t0 in set_joint_tuples
8:      while 1 do
9:          while each t in set_joint_tuples do
10:             sjtp = Probe_parent_nodes (t, sjtp, MX)
11:         end while
12:         if sjtp is empty then
13:             break;
14:         else
15:             set_joint_tuples = sjtp
16:             clear sjtp
17:         end if
18:     end while
19: end if
```

Function: `Probe_child_nodes` **(**t**,** MX**)**

```
1:  joinable_to_child = false
2:  while Each child nodes do
3:      while Each sub-space, sp, in WR (and N if child node is leaf node) do
4:          if cn_active not empty then
5:              while Each tuple t1 in sp joinable with t do
6:                  joinable_to_child = true
7:                  Matched_CN (cn_active, MX)
8:              end while
9:          end if
10:     end while
11: end while
12: Return joinable_to_child
```

Function: `Probe_parent_nodes` **(**t**,** $sjtp$**,** MX**)**

```
1:  while Each parent node, pn do
2:      if cn_active not empty then
3:          while Each tuple t1 in pn joinable with t do
4:              Matched_CN (cn_active, MX)
5:              put t1 in sjtp
6:          end while
7:      end if
8:  end while
9:  Return sjtp
```

Function: `Matched_CN` **(**cn_{active}**,** MX**)**

```
1:  Compute cn_newsubspace
2:  while Update branch_map of each CN in cn_active do
3:      if All bits in branch_map set to 1 then
4:          if Parent node is root node then
5:              Return all matched tuples (MTJNT) as result
6:              Update cn_newsubspace to fully match to ~CN.
7:          end if
8:          Move all matched tuples into appropriate sub-space cn_newsubspace
9:      end if
10: end while
```

Table 5.1: Parameters used in the experiments.

Parameter	Range and default
Window size (mn)	10, 20, **30**, 40, 50
Keyword frequency (%)	0.003, **0.007**, 0.01, 0.013
# of keywords	2, **3**, 4, 5
T_{max}	2, 3, **4**, 5

5.1.5 Discussion

In this section we elaborate the reason why the proposed scheme is advantageous to the existing approaches, S-KWS and SS-KWS. As we observed, the number of CNs exponentially increases as query keywords and/or T_{max} grows. Consequently, even though S-KWS and SS-KWS try to merge the CNs by finding common sub-networks, the size of query plans rapidly grows, which means a large number of CNs cannot share processing and need to be evaluated independently. Such redundant evaluation is very costly because it requires to scan all related tuples and check if they are joinable. This leads to very poor performance.

In MX-structure, we combine all CNs by consolidating all common edges without any restriction of node position. Thus, we can avoid the exponential blow up in the query plan, which means more CNs having overlapping edges can share processing. We enable MX-structure by keeping track of matching status using sub-spaces in each node buffer. It is true that the management of the complicated sub-buffers is not cost-free; however, that cost is very trivial comparing to that of independent evaluation of all unconsolidated CNs (very costly operation as explained above) in the query plans of S-KWS and SS-KWS. This leads to much better performance. We confirm this in the following experimental evaluation.

5.2 Experimental Evaluation

5.2.1 Setup and Datasets

SS-KWS [61], full mesh (FM) and partial mesh (PM) of S-KWS [59], and our proposed algorithm were implemented by using C++. All data structures and temporary data were entirely kept in the memory. All experiments were performed in Intel Core i7 CPU 870 @ 2.93GHz x 8 computer with 31.4 GiB memory in Ubuntu 13.10 (64 bits).

We used two types of datasets, synthetic and real datasets. For synthetic dataset, we used TPC-H dataset [57], which is about the transactions between customers and products. It is mainly used for testing performance of commercial DBMSs. In this dataset, there are 8 tables and 61 fields. Due to lack of real data stream datasets, we simulated DBLP [56], published in 2015, so that we could work on it as we work on real relational streams. The simulation was done by attaching time stamp to each tuple in DBLP dataset. And the simulator read tuples in the order defined by their time stamps and sent tuples continuously to the filtering system. DBLP dataset has 4 tables and 11 fields.

As explained earlier, SS-KWS performs better than FM and PM of S-KWS when the tuples coming from relational data streams mostly match CNs that have common edges at leaf nodes, at which lots of processings can be shared among those matched CNs. Therefore,

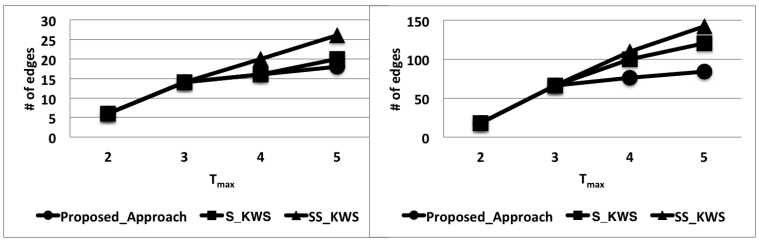
for experimental purposes, we separately prepare 2 datasets, one of which gives advantage to SS-KWS and the other gives advantage to S-KWS. Then, we investigate how the proposed algorithm can handle both kinds of datasets.

Parameters used in the experiment are shown in Table 5.1. We varied these parameters and compared the performance of the proposed algorithm with comparative algorithms, SS-KWS and PM/FM of S-KWS. The default parameters are written in bold.

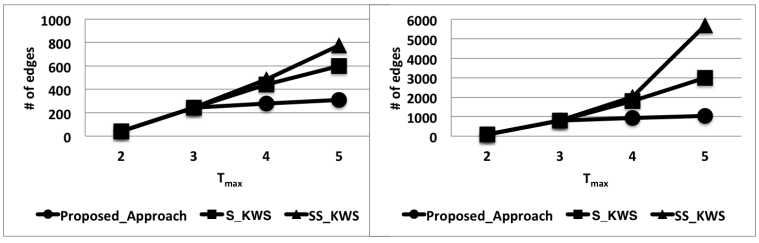
5.2.2 Comparison of Query Plans' Size

We first made a comparison of query plans' size (in terms of number of edges) of all approaches because they have great impact on the performance. For this experiment, we only used two parameters, number of query keywords and T_{max}, because other parameters do not have any impact on the size of query plan. We varied the number of query's keywords and T_{max} from 2 to 5 and investigated the increase in number of edges of all approaches.

The results are shown in Figures 5.7 and 5.8 for DBLP and TPCH respectively. As can be seen, when the number of query's keywords and T_{max} are increased, the total number of edges of all approaches increases for both datasets. We notice that there is an exponential increase of the number of edges in SS-KWS and S-KWS, which was caused by the explosion of number of CNs whose edges could not be consolidated in their query plans. Such explosive increase in size of query plans indicates that the performance of S-KWS and SS-KWS will greatly degrade when the number of query keywords and T_{max} increase. However, the growing rate of the proposed scheme was linearly increased because it consolidated unique edges into one, and the total number of unique edges, which were the primary/foreign-key relationships between two tables in the schema (which is usually comparatively small), in all CNs was slightly increased as the number of CNs increased. This proves that the proposed scheme can scale well with the increase in number of query keywords and T_{max}.

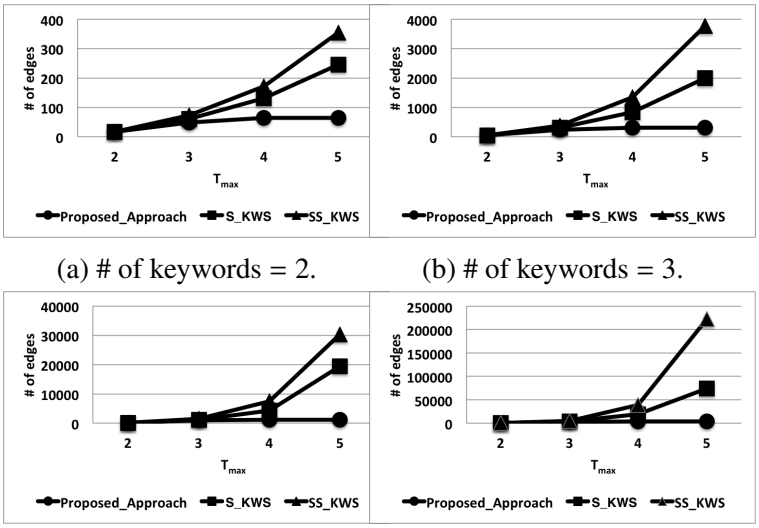


(a) # of keywords = 2. (b) # of keywords = 3.



(c) # of keywords = 4. (d) # of keywords = 5.

Figure 5.7: DBLP dataset: Comparison of number of edges



(a) # of keywords = 2. (b) # of keywords = 3.

(c) # of keywords = 4. (d) # of keywords = 5.

Figure 5.8: TPCH dataset: Comparison of number of edges

5.2.3 Performance Comparison

Dataset Giving Advantage to SS-KWS

This experiment was done on the datasets of DBLP [56] and TPCH [57] specially prepared so that SS-KWS outperforms S-KWS. We compared CPU running time, memory usage, and total number of probings. The results for DBLP are shown in Figures 5.9, 5.10, 5.11 and 5.12. Figures 5.13, 5.14, 5.15 and 5.16 show the results of TPCH dataset.

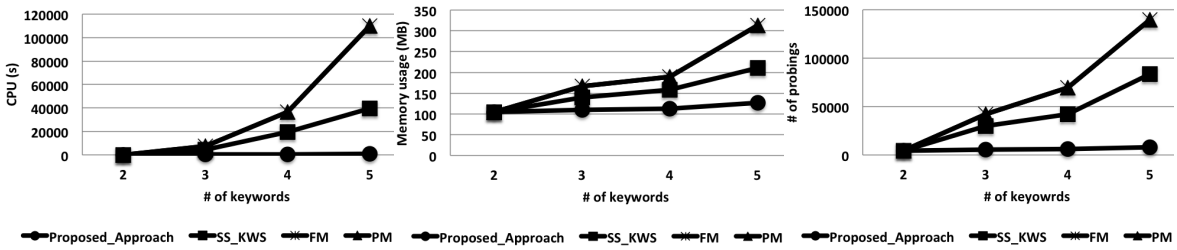
First, we measured the CPU running time and the memory usage when varying the number of keywords (Figures 5.9(a) and 5.9(b) for DBLP and Figures 5.13(a) and 5.13(b) for TPCH). As can be seen, for both datasets, CPU running time and the memory usage in FM/PM and SS-KWS were increased exponentially, whereas the proposed scheme was not. As an evidence, the number of probings was also exponentially increased in FM/PM and SS-KWS as shown in Figures 5.9(c) for DBLP and 5.13(c) for TPCH. This is due to the explosion in size of the query plans of FM/PM and SS-KWS as explained in the above experiment. Similar tendency can be observed when varying T_{max} from two to five (Figures 5.10 and 5.14).

Next, we increased the size of window from 10 min, 20 min, 30 min, 40 min, and 50 min. As expected, when the size of window was increased, the CPU running time, memory usage, and number of probings of all approaches also increased as shown in Figures 5.11 and 5.15 for DBLP and TPCH respectively. This was because fewer tuples in the buffers of all approaches were expired and deleted as a result of the increase in size of window. Figures 5.12 (DBLP) and 5.16 (TPCH) shows the impact on the performance of all approaches when varying keyword frequency. When keyword frequency was increased, there were more tuples containing the keywords of the query. As a result, there were more tuples that need to be joint. Therefore, the CPU running time, memory usage, and number of probings of all approaches also increased. Nevertheless, the total number of CNs did not increase when increasing window size and keyword frequency. Therefore, there is no change in size of query plans of all approaches, which caused little impact on the performance.

Dataset Giving Advantage to S-KWS

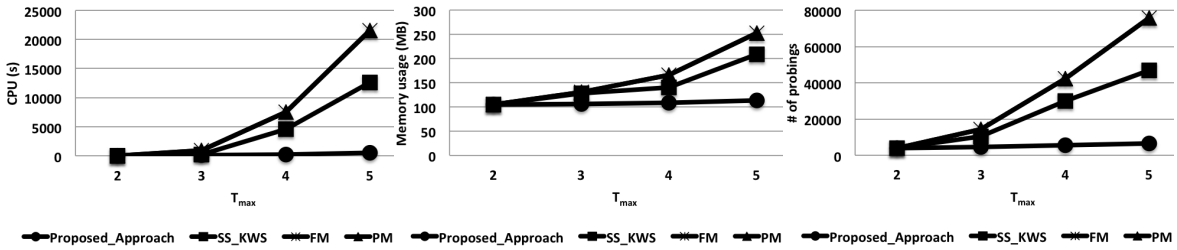
Next experiment was done on the relational streams of DBLP and TPCH, from which datasets were prepared to favor FM and PM of S-KWS. As shown in Figure 5.7 for DBLP and Figure 5.8 for TPCH, the number of edges that appears in lattice of SS-KWS is more than that in operator mesh of S-KWS. Therefore, by default, FM and PM of S-KWS perform better than SS-KWS. The trend is similar to that in the above experiment.

The results are shown in Figures 5.17, 5.18, 5.19 and 5.20 for DBLP. Figures 5.21, 5.22, 5.23 and 5.24 show the results of TPCH dataset. As can be seen, the results are similar to the above experiments that the proposed scheme greatly outperforms the existing approaches for all experimental parameters. Notice that, FM/PM of S-KWS outperforms SS-KWS for this dataset.



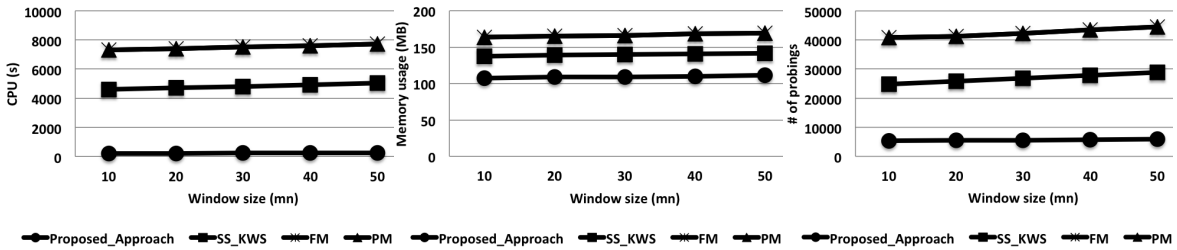
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.9: DBLP dataset (advantageous to SS-KWS): Varying # of keywords



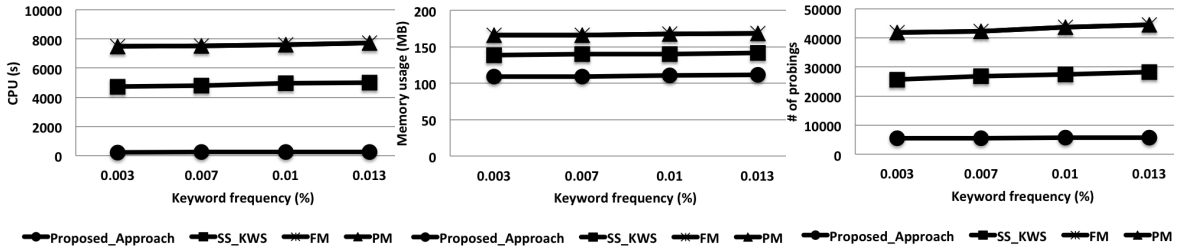
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.10: DBLP dataset (advantageous to SS-KWS): Varying T_{max}



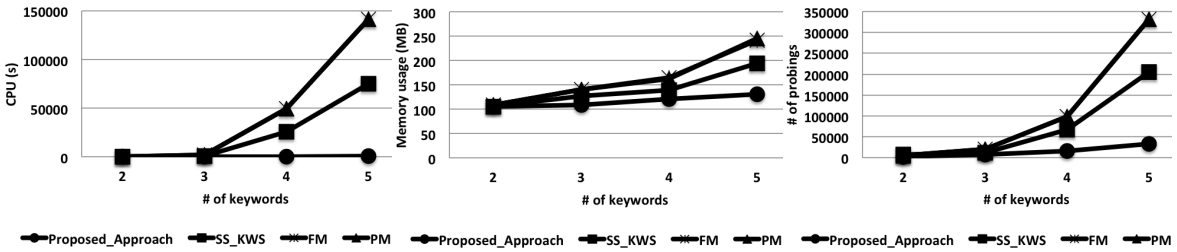
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.11: DBLP dataset (advantageous to SS-KWS): Varying window size



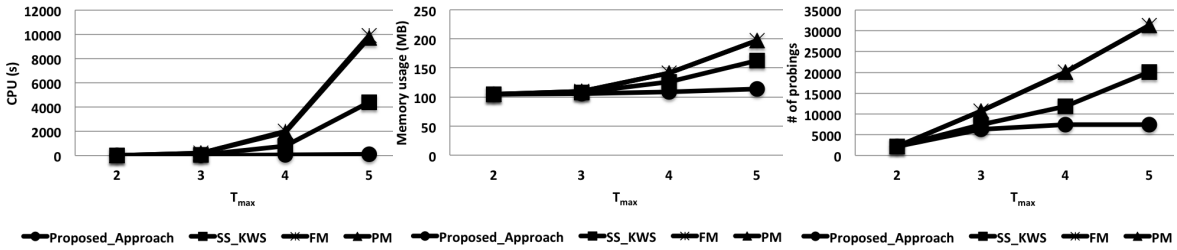
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.12: DBLP dataset (advantageous to SS-KWS): Varying keyword frequency



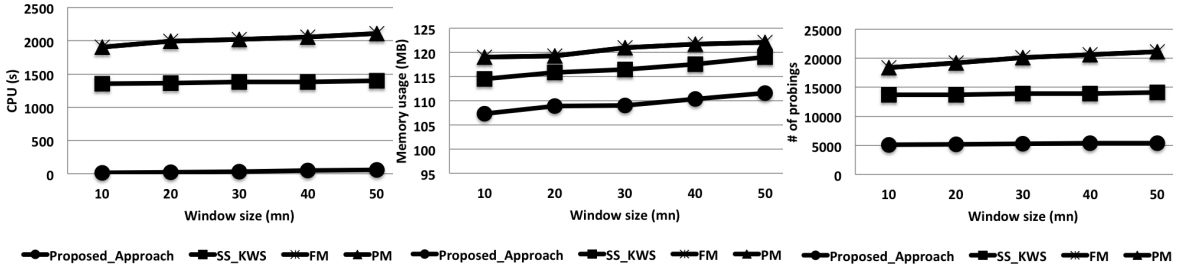
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.13: TPCH dataset (advantageous to SS-KWS): Varying # of keywords



(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.14: TPCH dataset (advantageous to SS-KWS): Varying T_{max}

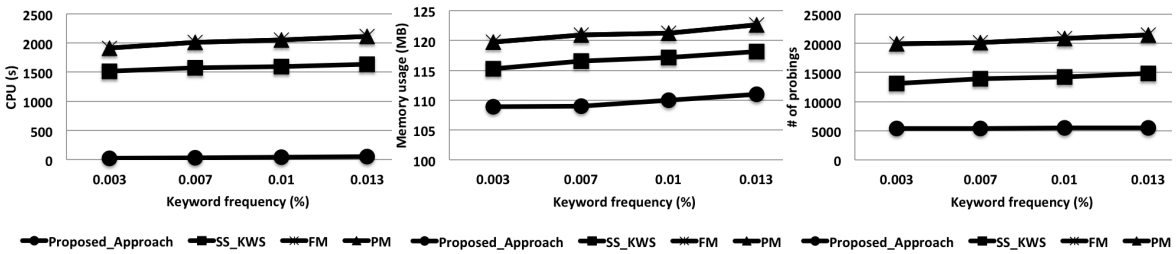


(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.15: TPCH dataset (advantageous to SS-KWS): Varying window size

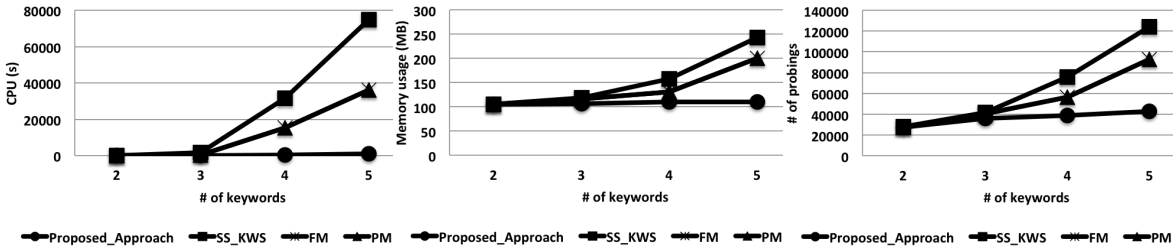
5.3 Summary of this Chapter

In this chapter we have proposed an improved method of keyword search over relational streams. In the proposed scheme candidate networks are merged into a novel data structure called MX-structure, and keyword search is efficiently processed based on the proposed



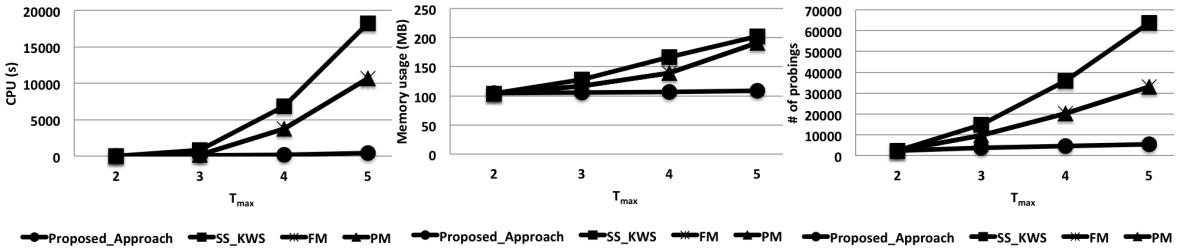
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.16: TPCH dataset (advantageous to SS-KWS): Varying keyword frequency



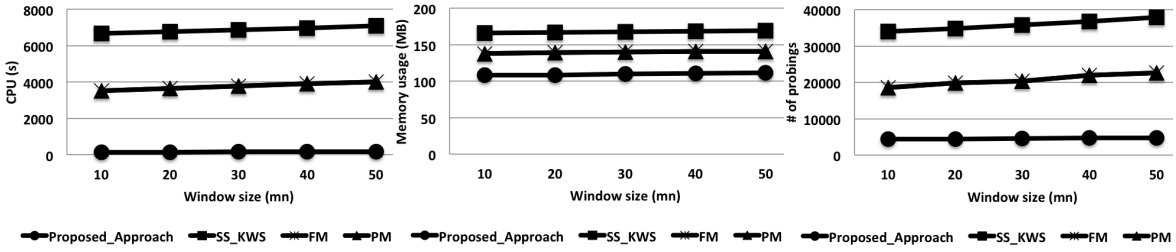
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.17: DBLP dataset (advantageous to S-KWS): Varying # of keywords



(a) CPU running times (b) Memory usage (c) # of probings

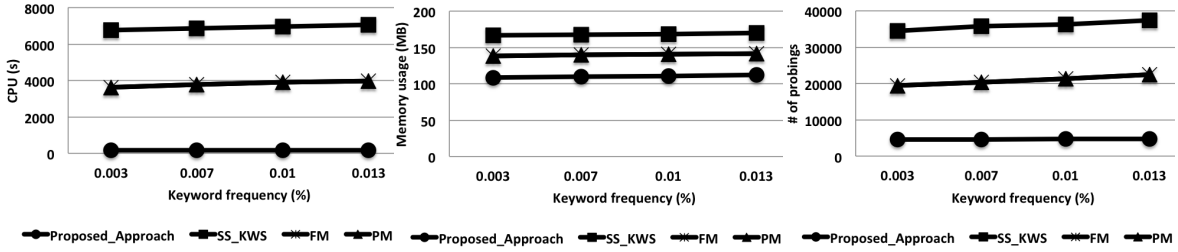
Figure 5.18: DBLP dataset (advantageous to S-KWS): Varying T_{max}



(a) CPU running times (b) Memory usage (c) # of probings

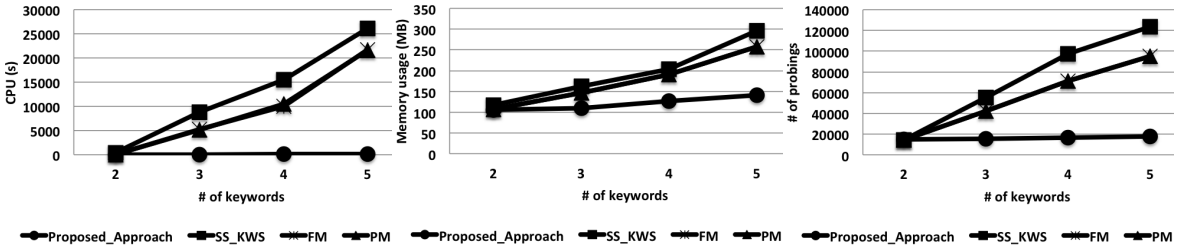
Figure 5.19: DBLP dataset (advantageous to S-KWS): Varying window size

algorithms with the help of MX-structure. The experimental results on both synthetic and real datasets have shown that the proposed scheme significantly outperforms the comparative methods even when the number of query keywords and/or T_{max} are increased.



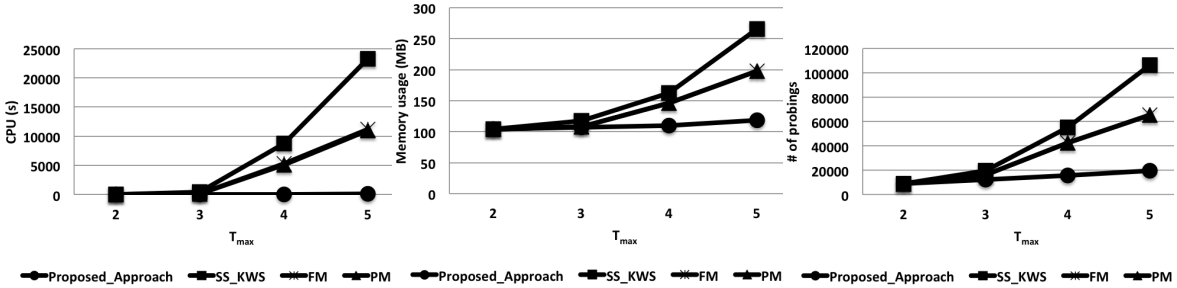
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.20: DBLP dataset (advantageous to S-KWS): Varying keyword frequency



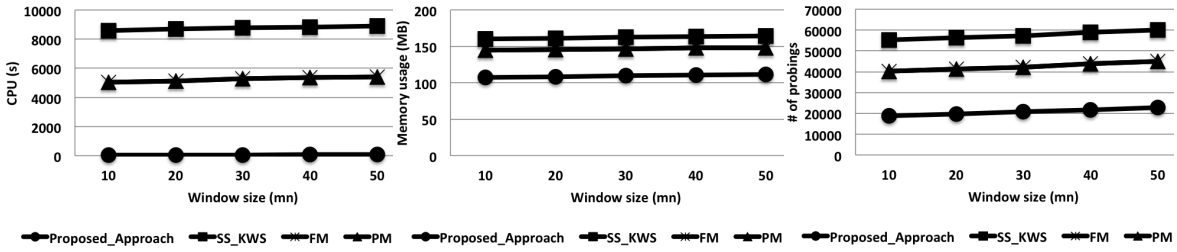
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.21: TPCH dataset (advantageous to S-KWS): Varying # of keywords



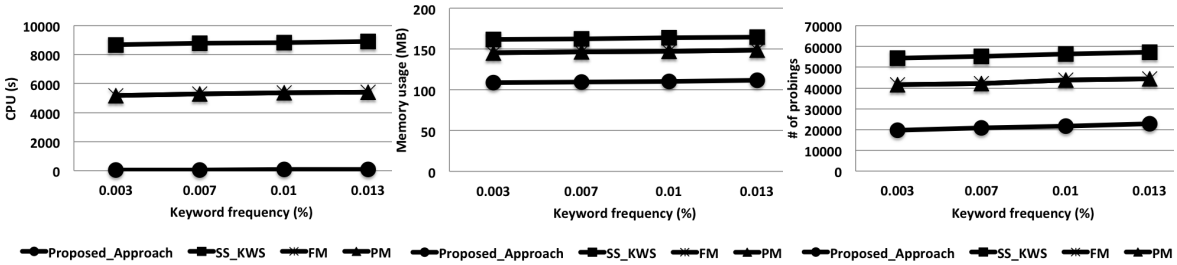
(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.22: TPCH dataset (advantageous to S-KWS): Varying T_{max}



(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.23: TPCH dataset (advantageous to S-KWS): Varying window size



(a) CPU running times (b) Memory usage (c) # of probings

Figure 5.24: TPCH dataset (advantageous to S-KWS): Varying keyword frequency

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this dissertation, we have studied the problem of keyword search over structured and semi-structured data streams. Specifically, we have focused on XML and relational streams because they are two popular data representations that have been extensively used in many applications for a long time. We have proposed two approaches regarding this search framework. They are XPath-based keyword search over XML streams and efficient method of keyword search over relational streams. We would like to summarize each contribution as follow.

6.1.1 XPath-based Keyword Search over XML Streams

For the first contribution, we have proposed XPath-based keyword search, that allows users to specify their search intentions more effectively. We have also proposed a mechanism to process this query over XML streams. Different from the existing algorithms that only support one specific type of query, our proposed method supports XPath, keyword search, and XPath-based keyword search. For this purpose, the NFA model is extended so that it supports XPath-based keyword search. 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 have evaluated the proposed approach by some experiments on both synthetic and real datasets. We investigated how the increase in number of queries and keywords in each query affects the performance of our proposed work. Through the experimental results, it has been proved that our proposed method works well with acceptable throughputs, less memory usage, and good efficiency and utility. In addition, our method gives a remarkable contribution to reduce the vagueness of keyword search; as a consequence, the level of relatedness of search's results is greatly improved, and the search results are more desirable. Moreover, CKStream only supports pure keyword search and YFilter only supports pure XPath, but the proposed approach supports more varieties of queries' types (XPath, keyword search, and XPath-based keyword search). More importantly, the proposed approach can achieve comparable throughputs to CKStream when processing pure keyword search, and comparable throughputs to YFilter when processing pure XPath. This proves that our proposed approach

is more practical in real world scenario of search engine.

6.1.2 Keyword Search over Relational Streams

For the second contribution, we have proposed an improved method of keyword search over relational streams. In the proposed scheme, candidate networks are merged into a novel data structure called MX-structure, and keyword search is efficiently processed based on the proposed algorithm with the help of MX-structure.

To prove the effectiveness of the proposed approach, extensive experiments have been done on both synthetic and real datasets. A variety of parameters, such as number of query keywords, T_{max}, window size, and keyword frequency, are used to measure how they affect the efficiency of the proposed approach and the comparative approaches. The experimental results show that the proposed scheme significantly outperforms the comparative approaches with regards to any parameters. Experimental results also prove that the performance of the comparative approaches greatly degrades when the number of query keywords and/or T_{max} are increasing because their query plans become exponentially big in terms of number of edges, so their performances become inefficient. The proposed approach can scale very well with respect to any parameter, and in particular greatly outperforms the comparative approaches when the number of query keywords and/or T_{max} are increased. Therefore, our proposed approach is more suitable for real search engine.

6.2 Future Work

In this section, we would like to present some possible future research extensions of the two proposed approaches.

6.2.1 XPath-based Keyword Search over XML Streams

For future research direction, we are going to apply our proposed method, XPath-based keyword search, to multiple XML streams, in which useful information can be obtained only when different sources are combined. Due to the need to get real-time answers, this combination is to be done at real-time based on the XPath-based keyword search.

In addition, we are also going to explore over more features of XPath full-text queries, which are more expressive in getting more complicated information, and study their deployment in streaming environment.

6.2.2 Keyword Search over Relational Streams

In this work, we have noticed that CN-based approach has some limitations. In particular some CNs are not used due to the biased keyword distribution in relational streams. For the future work, we plan to exploit such locality to enhance the performance by generating and processing only CNs that can produce results.

In addition, currently, our proposed approach and the comparative approaches, which are CN-based approach, only support a single keyword search at a time over relational streams.

It is close to impossible to process multiple keyword searches at the same time by using the above CN-based approaches because of the explosive blow up of all CNs. This is important and we hope that the idea of our proposed approach can be used to explore other approaches that does not rely on CN with an attempt to enable the processing of multiple keyword search over relational streams.

Bibliography

[1] V. Hristidis and N. Koudas, "Keyword Proximity Search in XML Trees," *Journal of Knowledge and Data Engineering, IEEE Transactions*, vol. 18, no. 4, pp. 525–539, 2006.

[2] W3C, "Extensible Markup Language(XML)," *http://www.w3.org/XML/*, , 2016.

[3] W3C, "Extensible Markup Language(XML)," *http://www.w3.org/TR/html4/sgml*, , 2016.

[4] RSS, "RSS Feed Reader, Your Tool for Saving Time and Money," *RSS.com*, , 2016.

[5] ATOM, "Atom Syndication Format," *www.w3.org/wiki/Atom*, , 2016.

[6] SOAP, "Simple Object Access Protocol," *www.w3.org/TR/soap/*, , 2016.

[7] XHTML, "XHTML," *www.w3.org/TR/xhtml1/*, , 2016.

[8] msoffice, "Language Accessory Pack for Office 2016," *www.Office.com*, , 2016.

[9] OpenDocument, "Language Accessory Pack for Office ," *http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=66375*, , 2016.

[10] Apple, "Apple's iWork," *http://apple.com/pr/library/2005/01/11Apple-Unveils-iWork-05.html*, , 2016.

[11] J. Leif, "XMPP as MOM - Greater NOrdic MIddleware Symposium (GNOMIS)," *Oslo: University of Stockholm. Archived from the original (PDF)*, , 2011.

[12] H. Stacey, "Announcing .NET Framework 4.6.2," *.NET Blog. Microsoft*, , 2016.

[13] W3C, "XML DOM (Document Object Model)," *http://www.w3schools.com/xml/xml_parser.asp*, , 2016.

[14] Apache XML project, "Xerces Java Parser 1.2.3," *http://xml.apache.org/xerces-j/index.html*, , 2016.

[15] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[16] "CODASYL Data Model," *Encyclopedia of Database Systems*, pp. 396–396, 2009.

[17] "Information Management System," *http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/ibmims/.*

[18] "Sabre Global Distribution System (GDS)," *https://en.wikipedia.org/wiki/Sabre_(computer_system)*, retrieved in 2016.

[19] "Oracle Database," *https://www.oracle.com/database/index.html*, retrieved in 2016.

[20] "MySQL," *http://www.mysql.com/*, retrieved in 2016.

[21] "Microsoft SQL Server," *https://www.microsoft.com/en-us/sql-server/default.aspx*, retrieved in 2016.

[22] "PostgreSQL," *https://www.postgresql.org/*, retrieved in 2016.

[23] "IBM DB2," *www-01.ibm.com/software/data/db2*, retrieved in 2016.

[24] "Structured Query Language (SQL)," *https://msdn.microsoft.com/en-gb/library/windows/desktop/ms714670(v=vs.85).aspx*, retrieved in 2016.

[25] J. Green, A. Gupta, and M. Onozuka, "Processing XML Stream with Deterministic Automata and Stream Indexes," *Journal of ACM Transaction and Database System*, vol. 29, no. 4, pp. 752–788, 2004.

[26] C. Hummel, S. da Silva, M. Moro, and H.F., Laender, "Multiple Keyword-based Queries over XML Streams," *Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM)*, pp. 1577–1582, 2011.

[27] Z. Vagena and M. Moro, "Semantic Search over XML Document Streams," *Proceedings of DATAX*, 2008.

[28] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents," *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 16–27, 2003.

[29] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "XSEarch: A Semantic Search Engine for XML," *Proceedings of the 29th international conference on Very large data bases*, vol. 29, pp. 45–56, 2003.

[30] S. Amer-Yahia, E. Curtmola, and A. Deutsch, "Flexible and Efficient XML Search with Complex Full-Text Predicates," *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 575–586, 2006.

[31] Z. Vagena, L. S. Colby, F. Ozcan, A. Balmin, and Q. Li, "On the Effectiveness of Flexible Querying Heuristics for XML Data," *Journal of Database and XMLTechnologies*, vol. 4704, pp. 77–91, 2007.

[32] Z. Liu and Y. Chen, "Identifying Meaningful Return Information for XML Keyword Search," *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 329–340, 2007.

[33] C. Sun, C.-Y. Chan, and A. K. Goenka, "Multiway SLCA-based Keyword Search in XML Data," *Proceedings of the 16th international conference on World Wide Web*, pp. 1043–1052, 2007.

[34] Y. Xu and Y. Papakonstantinou, "Efficient Keyword Search for Smallest LCAs in XML Databases," *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 527–538, 2005.

[35] Y. Diao and M. J. Franklin, "High-Performance XML Filtering: An Overview of YFilter," *IEEE Data Engineering Bulletin*, pp. 41–48, 2003.

[36] Y. Li, C. Yu, and H. V. Jagadish, "Schema-Free XQuery," *Proceedings of the Thirtieth international conference on Very large data bases (VLDB)*, vol. 30, pp. 72–83, 2004.

[37] C. Yu and L. Popa, "Constraint-based XML Query Rewriting for Data Integration," *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 371–382, 2004.

[38] 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.

[39] "World Geographic Database," *www/repository.html/mondial*, 2013.

[40] M. Gawinecki, F. Mandreoli, and G. Cabri, "Keyword Search over XML Streams: Addressing Time-Stamping and Understanding Results," *University of Modena*, pp. 371–382, 2008.

[41] M. Onizuka, "Light-weight XPath Processing of XML Stream with Deterministic Automata," *Proceedings of the twelfth international conference on Information and knowledge management (CIKM)*, pp. 342–349, 2003.

[42] B. Q. Truong, S. S Bhowmick, C. Dyreson, and A. Sun, "MESSIAH: Missing Element-Conscious SLCA Nodes Search in XML Data," *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 37–48, 2013.

[43] Y. Li, C. Yu, and H. V. Jagadish, "Exploiting the Relationship between Keywords for Effective XML Keyword Search," *Advances in Databases and Information Systems*, vol. 8133, pp. 232–245, 2013.

[44] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu, "XMLTK: An XML Toolkit for Scalable XML Stream Processing," *University of Washington, NTT*, 2003.

[45] H. Jiawei, C. Yixin, D. Guozhu, P. Jian, W. W. Benjamin, W. Jianyong, and Y. Dora, "Stream Cube: An Architecture for Multi-Dimensional Analysis of Data Streams," *Distributed and Parallel Databases*, vol. 18, pp. 173–197, 2005.

[46] R. Praveen and M. Bongki, "SketchTree: Approximate Tree Pattern Counts over Streaming Labeled Trees," *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pp. 80, 2006.

[47] B. Albert and G. Ricard, "Mining Adaptively Frequent Closed Unlabeled Rooted Trees in Data Streams," *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pp. 34–42, 2008.

[48] S. B. Kevin, E. Vuk, G. Rainer, B. Andrey, E. Mohamaed, K. Carl-Christian, O. Fatma, and J. S. Eugene, "Jaql: A Scripting Language for Large Scale Semistructured Data Analysis," *Proceedings of the VLDB Endowment*, vol. 4, pp. 1272–1283, 2011.

[49] Z. Xin, T. Hetal, and Z. Carlo, "Unifying the Processing of XML Streams and Relational Data Streams," *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pp. 50, 2006.

[50] B. Albert and G. Ricard, "Adaptive XML Tree Classification on eEvolving Data Streams," *Journal of Machine Learning and Knowledge Discovery in Databases*, vol. 5781, pp. 147–162, 2009.

[51] L. Wei, C. Kenneth, and P. Yinfei, "A Parallel Approach to XML Parsing," *Proceedings of the 7th IEEE/ACM International Conference on Grid*, vol. 5781, pp. 223–230, 2006.

[52] K. Jaehoon, K. Youngsoo, and Seog P., "PosFilter: An Efficient Filtering Technique of XML Documents Based on Postfix Sharing," *Proceedings of the 24th British national conference on Databases*, vol. 4587, pp. 70–81, 2007.

[53] K.S. Candan, W. Hsiung, S. Chen, J. Tatemura, and D. Agrawal, "AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering," *Proceedings of the 32nd International Conference on Very Large Data Bases*, pp. 559–570, 2006.

[54] D. Shaul, E. Gadi, G. Shai, and P. Eran, "DTL's DataSpot: Database Exploration Using Plain Language," *Proceedings of the 24rd International Conference on Very Large Data Bases*, pp. 645–649, 1998.

[55] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, U. Srivastava, and J. Widom, "STREAM: The Stanford Data Stream Management System," *Technical Report , Stanford InfoLab, http://ilpubs.stanford.edu:8090/641/*, 2004.

[56] "Computer Science Bibliography," *http://dblp.uni-trier.de/xml/*, 2015.

[57] "TPC-H Benchmark Dataset," *http://www.tpc.org/tpch/*, 2015.

[58] V. Hristidis and Y. Papakonstantinou, "DISCOVER: Keyword Search in Relational Databases," *Proceedings of the 28th international conference on Very Large Data Bases (VLDB)*, pp. 670–681, 2002.

[59] A. Markowetz, Y. Yang, and D. Papadias, "Keyword Search on Relational Data Streams," *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 605–616, 2007.

[60] A. Markowetz, Y. Yang, and D. Papadias, "Keyword Search over Relational Tables and Streams," *ACM Transactions on Database Systems (TODS)* , vol. 34, no. 3, pp. 1–50, 2009.

[61] L. Qin, J. Xu Yu, and L. Chang, "Scalable Keyword Search on Large Data Streams," *VLDB Journal*, vol. 20, no. 1, pp. 35–57, 2011.

[62] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: A System for Keyword-based Search over Relational Databases," *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pp. 5, 2002.

[63] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient IR-Style Keyword Search over Relational Databases," *Proceedings of the 29th International Conference on Very Large Data Bases*, vol. 29, pp. 850–861, 2003.

[64] H. He, H. Wang, Y. Wang, and X. Zhou, "BLINKS: Ranked Keyword Searches on Graphs," *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 305–316, 2007.

[65] Y. Luo, X. Lin, and W. P.S., "Spark: Top-k Keyword Query in Relational Databases," *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 115–126, 2007.

[66] V. Hristidis and Y. Papakonstantinou, "Keyword Search over Relational Databases: A Metadata Approach," *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 565–576, 2011.

[67] A. Arasu, S. Babu, and J. Widom, "CQL: A Language for Continuous Queries over Streams and Relations," *Workshop, DBPL 2003*, pp. 1–19, 2003.

[68] S. Carl and R. Carolyn, "A Relational Approach to Querying Streams," *IEEE Transactions*, vol. 2, no. 4, pp. 401–409, 1990.

[69] D. J. Abadi, D. Carney, U. Centinternel, M. Cherniack, C. Convery, S. Lee, M. Stonebraker, N. Tatbul, and S.B. Zdonik, "Aurora: A new Model and Architecture for Data Stream Management," *VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[70] A. Balmin, V. Hristidis, and Y. Papakonstantinou, "ObjectRank: Authority-based Keyword Search in Databases," *Proceedings of the 30th VLDB Conference*, pp. 564–575, 2004.

[71] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum, "Probabilistic Ranking of Database Query Results," *Proceedings of the Thirtieth international conference on Very large data bases (VLDB)*, vol. 30, pp. 888–899, 2004.

[72] A. Hulgeri, B. Bhalotia, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword Search in Databases," *IEEE Data Engineering Bulletin*, vol. 24, no. 3, pp. 22–32, 2001.

[73] F. Liu, C. Yu, W. Meng, and A. Chowdhury, "Effective Keyword Search in Relational Databases," *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 563–574, 2006.

[74] Q. Su and J. Widom, "Indexing Relational Database Content Offline for Efficient Keyword-based Search," *Proceedings of the 9th International Database Engineering & Application Symposium*, pp. 297–306, 2005.

[75] O. Pericles, S. Altigran, and M. Edleno, "Ranking Candidate Networks of Relations to Improve Keyword Search over Relational Databases," *IEEE 31st International Conference on Data Engineering (ICDE)*, pp. 399–410, 2015.

[76] K. Mehdi, A. Aijun, C. Nick, G. Parke, S. Jaroslaw, and Y. Xiaohui, "Meaningful Keyword Search in Relational Databases with Large and Complex Schema," *IEEE 31st International Conference on Data Engineering (ICDE)*, pp. 411–422, 2015.

[77] Z. Zhong, B. Zhifeng, L. Mong, and L. Tok, "Towards An Interactive Keyword Search over Relational Databases," *WWW journal (companion volume)*, vol. 24, no. 3, pp. 259–262, 2015.

[78] K. Mehdi, A. Aijun, C. Nick, G. Parke, S. Jaroslaw, and Y. Xiaohui, "MeanKS: Meaningful Keyword Search in Relational Databases with Complex Schema," *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 905–908, 2014.

[79] X. Yanwei, G. Jihong, and I. Yoshiharu, "Scalable Top-k Keyword Search in Relational Databases," *Proceedings of the 17th international conference on Database Systems for Advanced Applications (DASFAA)*, pp. 65–80, 2012.

[80] S. Bou, T. Amagasa, and H. Kitagawa, "Filtering XML Streams by XPath and Keywords," *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pp. 410–419, 2014.

[81] F. Hummel, A. Silva, M. Moro, and A. Laender, "Multiple Keyword-based Queries over XML Streams," *Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM)*, pp. 1577–1582, 2011.

[82] S. Elbassuoni and R. Blanco, "Keyword Search over RDF Graphs," *Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM)*, pp. 237–242, 2011.

[83] X. Lian, L. Chen, and Z. Huang, "Keyword Search Over Probabilistic RDF Graphs," *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING (TKDE)*, vol. 27, no. 5, pp. 1246–1260, 2015.

[84] L. Edward, "Cyber Physical Systems: Design Challenges," *University of California, Berkeley Technical Report No. UCB/EECS-2008-8. Retrieved 2008-06-07*, 2008.

[85] O. Niggermann and V. Lohweg, "On the Diagnosis of Cyber-Physical Production Systems," *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI)*, pp. 4119–4126, 2015.

[86] M. Dyk, A. Najgebauer, and D. Pierzchala, "Agent-Based MS of Smart Sensors for Knowledge Acquisition Inside the Internet of Things and Sensor Networks," *Asian Conference on Intelligent Information and Database Systems (ACIIDS)*, vol. 9012, pp. 224–234, 2015.

[87] H. Zhang, C. Sanin, and E. Szczerbicki, "Experience-Oriented Enhancement of Smartness For Internet of Things," *Asian Conference on Intelligent Information and Database Systems (ACIIDS)*, vol. 9012, pp. 506–515, 2015.

[88] K. Hogan, "Interpreting Hitwise Statistics on Longer Queries," *Technicall report, Ask.com*, 2009.

[89] A. Arasu, S. Babu, and J. Widom "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *The International Journal on Very Large Data Bases* , vol. 15, no. 2, pp. 121–142, 2006.

[90] A. Arasu, S. Babu, and J. Widom "CQL: A Language for Continuous Queries over Streams and Relations," *International Workshop on Database Programming Languages (DBPL)*, vol. 2921, pp. 1–19, 2003.

[91] A. Arasu, S. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom "STREAM: The Stanford Data Stream Management System," *Data Stream Management Part of the series Data-Centric Systems and Applications*, pp. 317–336, 2016.

[92] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik "Aurora: A New Model and Architecture for Data Stream Management," *The International Journal on Very Large Data* , vol. 12, no. 2, pp. 120–139, 2003.

[93] M. Ray, C. Lei, and E. A. Rundensteiner "Scalable Pattern Sharing on Event Streams," *Proceedings of the 2016 International Conference on Management of Data* , pp. 495–510, 2016.

[94] S. Pan and X. Zhu "CGStream: Continuous Correlated Graph Query for Data Streams," *Proceedings of the 21st ACM international conference on Information and knowledge management* , pp. 1183–1192, 2012.

[95] N. Cipriani, O. Schiller, and B. Mitschang "M-TOP: Multi-target Operator Placement of Query Graphs for Data Streams," *Proceedings of the 15th Symposium on International Database Engineering & Applications* , pp. 52–60, 2011.

[96] M. Daum, F. Lauterwald, P. Baumgartel, and K. Meyer-Wegener "Propagation of Densities of Streaming Data within Query Graphs," *International Conference on Scientific and Statistical Database Management (SSDBM)* , pp. 584–601, 2010.

[97] E. Kwan, P. Hsu, J. Liang, and Y. Chen "Event Identification for Social Streams Using Keyword-based Evolving Graph Sequences," *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp. 450–457, 2013.

[98] M. S. Hassan, W. G. Aref, and A. M. Aly "Event Identification for Social Streams Using Keyword-based Evolving Graph Sequences," *Proceedings of the 2016 International Conference on Management of Data*, pp. 1183–1197, 2016.

[99] E. G. Barros, A. H.F. Laender, M. M. Moro, and A. S. Silva "LCA-based Algorithms for Efficiently Processing Multiple Keyword Queries over XML Streams," *Journal of Data & Knowledge Engineering*, vol. 103, pp. 1–18, 2016.

[100] E. G. Barros, F. G. D. C. Ferreira, and A. H. F. Laender "Parallelizing Multiple Keyword Queries over XML Streams," *2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW)*, pp. 169–172, 2016.

[101] E. G. Barros, A. H. F. Laender, M. M. Moro, and A. S. Silva "MKStream: An Efficient Algorithm for Processing Multiple Keyword Queries over XML Streams," *International Conference on Conceptual Modeling*, pp. 100–107, 2014.

[102] W. Yang, F. Fang, N. Li, and J. Lu "XKFitler: A Keyword Filter on XML Stream," *International Journal of Information Retrieval Research (IJIRR)*, vol. 1, no. 1, pp. 1–18, 2011.

[103] L. Li, H. Wang, J. Li, and H. Gao "Efficient Algorithms for Skyline Top-K Keyword Queries on XML Streams," *International Conference on Database Systems for Advanced Applications*, pp. 283–287, 2009.

[104] L. Li, H. Wang, J. Li, and J. Luo "Efficient Top-k Keyword Search on XML Streams," *The 9th International Conference for Young Computer Scientists*, pp. 1041–1046, 2008.

[105] W. Yang and B. Shi "Schema-Aware Keyword Search over XML Streams," *7th IEEE International Conference on Computer and Information Technology (CIT)*, pp. 29–34, 2007.

[106] K. Abakumov, "JsFlow: Integration of Massive Streams and Batches via JSON-based Dataflow Algebra," *Proceedings of the Ninth Spring Researchers Colloquium on Database and Information Systems*, pp. 35–38, 2013.

[107] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "C-SPARQL: a Continuous Query Language for RDF Data Streams," *Int. J. Semantic Computing*, vol. 4, no. 1, pp. 3–25, 2010.

[108] D. F. Barbrieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "Querying RDF Streams with C-SPARQL," *ACM SIGMOD Record* , vol. 39, no. 1, pp. 20–26, 2010.

[109] S. Gillani, G. Picard, and F. Laforest, "DIONYSUS: Towards Query-aware Distributed Processing of RDF Graph Streams," *EDBT/ICDT Workshops*, 2016.

[110] S. Groppe, J. Groppe, D. Kukulenz, and V. Linnemann, "A SPARQL Engine for Streaming RDF Data," *Third International IEEE Conference on Signal-Image Technologies and Internet-Based System*, pp. 167–174, 2008.

[111] D. L. Phuoc, M. D. Tran, A. L. Tuan, M. N. Duc, and M. Hauswirth, "RDF Stream processing with CQELS Framework for Real-time Analysis," *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pp. 285–292, 2015.

[112] M. Wei, E. A. Rundensteiner, M. Mani, and M. Li, "Processing Recursive XQuery over XML Streams: The Raindrop Approach," *3rd XML Schema and Data Management Workshop*, vol. 65, no. 2, pp. 243–265, 2008.

[113] M. Wei, M. Li, E. A. Rundensteiner, and M. Mani, "Processing Recursive XQuery over XML Streams: The Raindrop Approach," *22nd International Conference on Data Engineering Workshops*, pp. 85, 2006.

[114] H. Su, E. A. Rundensteiner, and M. Mani, "Semantic Query Optimization for XQuery over XML Streams," *Proceedings of the 31st VLDB Conference*, pp. 277–288, 2005.

[115] H. Su, E. A. Rundensteiner, and M. Mani "Semantic Query Optimization in an Automata-Algebra Combined XQuery Engine over XML Streams," *Proceedings of the 30st VLDB Conference*, pp. 1293–1296, 2004.

[116] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier, "FluXQuery: An Optimizing XQuery Processor for Streaming XML Data," *Proceedings of the 30st VLDB Conference*, pp. 1309–1312, 2004.

[117] J. Jian, H. Su, and E. A. Rundensteiner, "Automaton Meets Query Algebra: Towards a Unified Model for XQuery Evaluation over XML Data Streams," *International Conference on Conceptual Modeling*, pp. 172–185, 2003.

[118] W. H. Tok, S. Bressan, and M. Lee, "Twig'n Join: Progressive Query Processing of Multiple XML Streams," *International Conference on Database Systems for Advanced Applications*, pp. 546–553, 2008.

[119] C. Chou, K. Jea, and H. Liao, "A Syntactic Approach to Twig-query Matching on XML Streams," *Journal of Systems and Software*, vol. 84, no. 6, pp. 993–1007, 2011.

[120] J. P. Park, C. Park, and Y. D. Chung, "Lineage Encoding: An Efficient Wireless XML Streaming Supporting Twig Pattern Queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 7, pp. 1559–1573, 2013.

[121] M. Ray, C. Lei, and E. A. Rundensteiner, "Scalable Pattern Sharing on Event Streams," *Proceedings of the 2016 International Conference on Management of Data*, pp. 495–510, 2016.

[122] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang, "AP-Tree: Efficiently Support Continuous Spatial-keyword Queries over Stream," *IEEE 31st International Conference on Data Engineering (ICDE)*, pp. 1107–1118, 2015.

[123] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang, "AP-Tree: Efficiently Support Location-aware Publish/Subscribe," *The VLDB Journal*, vol. 24, no. 6, pp. 823–848, 2015.

[124] L. Guo, D. Zhang, G. Li, K. Tan, and Z. Bao, "Location-Aware Pub/Sub System: When Continuous Moving Queries Meet Dynamic Event Streams," *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 843–857, 2015.

[125] H. Abdelhaq and M. Gertz, "On the Locality of Keywords in Twitter Streams," *Proceedings of the 5th ACM SIGSPATIAL International Workshop on GeoStreaming*, pp. 12–20, 2014.

[126] X. Wang, Y. Zhang, W. Zhang, and X. Lin, "Efficiently Identify Local Frequent Keyword Co-occurrence Patterns in Geo-tagged Twitter Stream," *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pp. 1215–1218, 2014.

[127] V. Hristidis, O. Valdivia, M. Vlachos, and P. S. Yu, "Continuous Keyword Search on Multiple Text Streams," *Proceedings of the 15th ACM international conference on Information and knowledge management Pages*, pp. 802–803, 2006.

[128] V. Hristidis, O. Valdivia, M. Vlachos, and P. S. Yu, "A System for Keyword Search on Textual Streams," *Proceedings of the 2007 SIAM International Conference on Data Mining*, pp. 503–508, 2007.

[129] D. J. Kuss and M. D. Griffiths, "Online Social Networking and Addiction-A Review of the Psychological Literature," *International Journal of Environmental Research and Public Health*, vol. 8, no. 9, pp. 3528–3552, 2011.

[130] Twitter, "http://twitter.com"

[131] Facebook, "http://facebook.com"

[132] Facebook, "http://linkedin.com"

[133] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, "STREAM: The Stanford Stream Data Manager," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 26, no. 1, pp. 19–26, 2003.

[134] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "STREAM: The Stanford Stream Data Manager," *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 665–665, 2003.

[135] SAP Event Stream Processor, "http://sap.com/uk/product/data-mgmt/complex-event-processing.html"

[136] SAP Event Stream Processing Engine, "http://sas.com/en_us/software/data-management/event-stream-processing.html"

[137] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams," *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 577–588, 2013.

[138] K. Nakao, K. Yoshioka, D. Inoue, and M. Eto, "A Novel Concept of Network Incident Analysis based on Multi-layer Observations of Malware Activities," *The 2nd Joint Workshop on Information Security (JWIS07)*, pp. 267–279, 2007.

[139] D. Inoue, M. Eto, K. Yoshioka, S. Baba, K. Suzui, J. Nakazato, K. Ohtaka, and K. Nakao, "nicter: An Incident Analysis System Toward Binding Network Monitoring with Malware Analysis," *WOMBAT Workshop on Information Security Threats Data Collection and Sharing (WISTDCS 2008)*, pp. 58–66, 2008.

[140] N. Junji and O. Kazuhiro, "nicter Report-Tansition Analysis of Cyber Attacks based on Long-term Observation," *Journal of the National Institute of Information and Communications Technology*, vol. 58, no. 3/4, pp. 27–34, 2011.

[141] K. Ohtake, J. Goto, S. D. Saeger, and K. Torisawa, "NICT Disaster Information Analysis System," *The Companion Volume of the Proceedings of IJCNLP 2013; System Demonstrations*, pp. 29–32, 2013.

[142] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-min Sketch and Its Applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[143] S. Ganguly, M. Bansal, and S. Dube, "Estimating Hybrid Frequency Moments of Data Streams," *Journal of Combinatorial Optimization*, vol. 23, no. 3, pp. 373–394, 2012.

[144] D. Wang, E. A. Rundensteiner, and R. T. Ellison, "Active Complex Event Processing over Event Streams," *Proceedings of the VLDB Endowment*, vol. 4, no. 10, pp. 634–645, 2011.

[145] Z. Feng, M. Wang, S. Yang, and L. Jiao, "Incremental Semi-Supervised Classification of Data Streams via Self-representative Selection," *Proceedings of the VLDB Endowment*, vol. 47, pp. 389–394, 2016.

[146] J. Liin, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, No gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams," *ACM SIGMOD Record*, vol. 34, no. 1, pp. 39–44, 2005.

[147] Y. Ji, J. Sun, A. Nica, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Quality-driven Disorder Handling for m-way Sliding Window Stream Joins," *IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 493–504, 2016.

[148] S. Halle and R. Villemaire, "Runtime Monitoring of Web Service Choreographies Using Streaming XML," *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 2118–2125, 2009.

[149] J. Paik, J. Nam, U. Kim, and D. Won, "Association Rule Extraction from XML Stream Data for Wireless Sensor Networks," *Journal of Sensors*, vol. 14, no. 7, pp. 12937–12957, 2014.

[150] J. Xu Yu, L. Qin, and L. Chang, "Keyword Search in Relational Databases: A Survey," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 33, no. 1, pp. 67–78, 2010.

[151] J. Lu, "Effective XML Keyword Search," *Book of An Introduction to XML Query Processing and Keyword Search*, pp. 203–232, 2013.

List of Publications

Publications related to the dissertation

Refereed journal papers

- Savong Bou, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "Path-Based Keyword Search over XML Streams," *International Journal of Web Information Systems (IJWIS)*, Vol. 11, Iss. 3, pp. 347–369, 2015.

Refereed international conference papers

- Savong Bou, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "An Improved Method of Keyword Search over Relational Data Streams by Aggressive Candidate Network Consolidation," in *Proceedings of the 27th International Conference on Database and Expert Systems Applications (DEXA '16)*, pp. 336–351, Porto, Portugal, Sept. 5–8, 2016.
- Savong Bou, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "Filtering XML Streams by XPath and Keywords," in *Proceedings of 16th International Conference on Information Integration and Web-based Applications & Services (iiWAS '14)*, pp. 410–419, Hanoi, Vietnam, Dec. 4–6, 2014.
- Savong Bou, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "Keyword Search with Path-Based Filtering over XML Streams," in *Proceedings of IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS '14)*, pp. 337–338, Nara, Japan, Oct. 6–9, 2014. (Poster presentation)

Non-refereed Domestic Workshop Papers

- Savong Bou, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "Efficient Keyword Search over Relational Data Streams," in *8th Forum on Data Engineering and Information Management (DEIM '16)*, A3-4, Feb. 29–Mar. 2, 2016.
- Savong Bou, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "An Improved Method of Keyword Search over Relational Data Streams," in *7th Forum on Data Engineering and Information Management (DEIM '15)*, A3-2, Mar. 2–4, 2015.
- Savong Bou, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "Path-Based Keyword Search over XML Streams," in *6th Forum on Data Engineering and Information Management (DEIM '14)*, A1-5, Mar. 3–5, 2014.